VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

DIPLOMOVÁ PRÁCE

SOMA v CUDA

SOMA in CUDA

2016 Bc. Boháč Petr

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Petr Boháč**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: SOMA v CUDA
SOMA in CUDA

Jazyk vypracování: čeština

Zásady pro vypracování:

SOMA je jedním z řady evolučních algoritmů, které jsou požívány v optimalizačních úlohách. Speciálním případem jsou permutační optimalizační problémy, pro které existuje diskrétní varianta uvedeného algoritmu pojmenovaná DSOMA. Cílem této diplomové práce je provést implementaci tohoto algoritmu pro prostředí GPU. V práci proveďte
následující:

1. Seznamte se s problematikou heuristických algoritmů pro řešení permutačních optimalizačních problémů, zaměřte se na algoritmus DSOMA.
2. Proveďte referenční implementaci algoritmu DSOMA pro CPU v jazyce C#.
3. Reimplementujte algoritmus z předchozího kroku pro GPU v prostředí CUDA 7.5 a zaměřte se na využití pokročilých technik, např. unifikovaný paměťový model apod.
4. Obě implementace ověřte na klasických plánovacích problémech.
5. Porovnejte efektivitu (doba běhu, škálovatelnost apod.) obou implementací na sadě vhodných testovacích úloh.
6. Vše pečlivě popište v textové části práce a zhodnoťte přínos GPU.

Seznam doporučené odborné literatury:

[1] Sanders, J and Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional. Boston, USA, 2010.
[2] Farber, R.: CUDA Application Design and Development. Morgan Kaufmann, USA, 2011.
[3] Zelinka, I.: SOMA – Self Organizing Migrating Algorithm. In: G. Onwubolu, B. V. Babu, New Optimization Techniques in Engineering, Springer-Verlag, 2004.
[4] Zelinka, I. and Lampinen, J.: Soma—self-organizing migrating algorithm, In: Mendel, 6th International Conference on Soft Computing, Brno, Czech Republic, 2000.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Tomáš Fabián, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 15.07.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, kterou jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Tímto bych chtěl poděkovat mému vedoucímu práce Ing. Tomáši Fabiánovi, Ph.D. za pomoc kdykoliv jsem potřeboval. Také chci poděkovat mojí rodině za trpělivost a podporu při studiu a mé přítelkyni Lucii za pomoc.

PETR BOHÁČ, bohcet

Jméno a příjmení, podpis

Abstrakt

Cílem této diplomové práce je implementace SOMA v CUDA. Přesněji její diskrétní verze nazvaná DSOMA. Vytvořil jsem referenční algoritmus DSOMA v jazyce C#, využívající paralelizace. Pomocí něj byla naimplementována další aplikace pro CUDA, která z ní vycházela a využívala všechny výhody programování na grafické kartě. Ve fázi testování jsem nejprve porovnal oba implementované algoritmy. Následně byla u jednoho z algoritmů, tedy v CUDA, ověřena efektivita kódu. Aplikace byla zkoušena na různě velkých problémech a následovalo hledání nejlepšího řešení daného problému. Ověřil jsem možnost záměny velikosti populace s počtem migrací. Nakonec jsem otestoval využití prostředků počítače při běhu aplikace. Testováním jsem zjistil, že již při velikosti populace 1000 jedinců a Jmin = 1 je CUDA aplikace rychlejší než aplikace C#. Dále jsem zjistil, že při využití větší populace a provedení malého počtu migrací dosahuje DSOMA lepších výsledků a v porovnání s C# dosahuje i vyšší rychlosti výpočtu.

Abstract

The topic of this thesis is implementation of SOMA in CUDA. More precisely, it is the discrete version called DSOMA. I created a reference DSOMA algorithm in C# using parallelization. Based on this algorithm I implemented additional application for CUDA, which used all benefits of programming on graphic card. In the testing phase I first compared both implemented algorithms. Subsequently I verified the effectiveness of a code in CUDA. The application has been tested on various problems, followed by searching for the best solution of the particular problem. I examined the possibility of confusion of population size with number of migrations. Finally, I tested the use of computer resources at runtime. I found out by testing that when size of population is bigger than 1000 and Jmin = 1, CUDA application is faster than C# application. I also discovered that when a bigger population is used and small number of migrations is performed, DSOMA achieves better results and in comparison with C# achieves a higher calculation speed.

Klíčová slova

Flow Shop, CUDA, Plánovací úlohy, SOMA

Keywords

Flow Shop, CUDA, Scheduling problems, SOMA

Obsah

Seznam obrázků, tabulek a kódu

1. Úvod

Nalézt řešení kombinatoricky optimalizačního problému není ve většině případů snadné. Metodou ověření všech možných řešení problému by nalezení řešení trvalo velmi dlouho, případně bychom se výsledku nikdy nedočkali. Tyto problémy jsou velice rozsáhlé a množina možných řešení je obrovská. Proto je potřeba zvolit správnou strategii, a když už nemáme možnost získat nejoptimálnější řešení, chceme znát řešení, které se k němu alespoň blíží.

Toho se dá dosáhnout mnoha způsoby a jednou z oblíbených variant jsou evoluční algoritmy. Tyto algoritmy napodobují procesy či chování, které byly vypozorovány v biologii, jako jsou dědičnost, mutace nebo křížení. Jedním z těchto algoritmů je Samo-Organizační Migrační Algoritmus, zkráceně SOMA. SOMA si získala uznání na poli hledání globálních extrémů spojitých funkcí a je konkurenceschopná vůči ostatním evolučním algoritmům. Algoritmus byl modifikován, aby řešil permutační optimalizační úlohy. Algoritmus pojmenovali diskrétní SOMA (DSOMA). I algoritmus v této verzi se ukázal jako velice efektivní a dosahoval skvělých výsledků.

Jeden ze zástupců permutačních optimalizačních úloh je plánovací úloha nazývána Flow Shop. Řešením problému je nalezení nejlepší permutace posloupnosti úloh, spuštěných na určitém počtu strojů.

Algoritmus DSOMA jsem nejprve naimplementoval v jazyce C#, ale nyní existuje i vhodnější jazyky pro implementaci. U evolučních algoritmů jako je DSOMA se pracuje nad populací jedinců, kde výpočty mohou probíhat paralelně. A tento přístup je v poslední době doménou grafických karet. Grafické karty už dávno nejsou pouze jednoúčelová zařízení, která slouží k vykreslení scény na monitor. V současné době mají tisíce programovatelných jader a lze na nich provádět masivní paralelizace úkolů pomocí vhodného programovacího jazyku. Tím nejvyspělejším je jazyk CUDA, který vyvíjí společnost NVIDIA.

Cílem této práce je naimplementovat algoritmus DSOMA v CUDA a porovnat ho s referenčním řešením v jazyce C#. Porovnat dobu běhu obou programů a ověřit škálovatelnost na permutačních optimalizačních problémech. Konkrétně na různě velkých problémech Flow Shop. Ověřit zaměnitelnost velikosti populace a počtu provedených migrací a využití prostředků počítače při výpočtu.

2. Optimalizační problémy

Optimalizační problémy jsou problémy hledání nejlepšího řešení ze všech dosažitelných řešení [1]. Optimalizační problémy jsou rozděleny do dvou kategorií, podle toho, jestli jsou proměnné kontinuální nebo diskrétní. Optimalizační problémy s diskrétní proměnnou jsou označovány jako Kombinačně optimalizační problémy. V tomto druhu problému hledáme objekt, jako je celé číslo, permutace nebo graf. Problémy s kontinuálními proměnnými zahrnují vázané a multimodální problémy.

2.1 Typy optimalizačních problémů

Optimalizační problémy dělíme do několika skupin. První jsou kontinuální optimalizační problémy. Typickým příkladem může byt hledání minimální hodnoty libovolné kontinuální funkce. Dalším případem je kombinačně optimalizační problém. Zde již hledáme kombinaci vstupních dat takovou, pro které lze vypočítat nejlepší řešení. Poslední skupinou jsou NP optimalizační problémy, která pouze zpřísňuje kritéria, aby byl problém vypočítán v polynomiálním čase [1].

2.1.1 Kontinuální optimalizační problémy

Obecný popis kontinuálního optimalizačního problému je popsán v rovnicích 2.1 [1]. Je zde i vyjádřeno co která funkce představuje.

$$\begin{aligned} &\text{minimalizuj } x \quad f(x) \\ &g_i(x) \leq 0, \quad i = 1, \dots, m \\ &h_i(x) = 0, \quad i = 1, \dots, p \end{aligned} \tag{2.1}$$

$f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ je funkce minimalizace skrz proměnnou x

$g_i(x) \leq 0$ je nazývána nerovnostním omezením

$h_i(x) = 0$ je nazýváno rovnostním omezením

Tento předpis, kde hledáme minimální řešení problému, můžeme předělat na hledání maximální řešení problému. To se provede tak, že znegujeme funkci f.

2.1.2 Kombinačně optimalizační problémy

Formálně je kombinačně optimalizační problém čtveřice, znázorněna v rovnici 2.2 níže [1]. Následuje i její podrobné vysvětlení.

$$A = (I. f. m. g), \tag{2.2}$$

Mějme funkci $f(x)$, kde x náleží I a I je množina všech řešení. Hledáme pak takové x, aby funkce $f(x)$ byla minimální nebo opačně maximální. Potom se dá hovořit o minimalizačním či maximalizačním problému. Hodnota m vyjadřuje dosažitelná řešení daného problému. Parametr g je cílová funkce, která vyjadřuje minimum nebo maximum.

Pro každý kombinačně optimalizační problém existuje odpovídající rozhodovací problém, který se ptá, jestli existuje nějaké dosažitelné řešení m pro částečné řešení daného problému. Pro příklad máme dva vrcholy a, b v grafu G. V tomto případě zadání optimalizačního problému může být následující. Nalezněte nejkratší cestu mezi vrcholy a a b, tak aby graf obsahoval co nejmenší počet hran. Odpovědí bude nějaké celočíselné řešení. Odpovídající rozhodovací problém potom může být tento. Je v grafu mezi vrcholy a a b cesta, který má méně jak 5 hran? Pro podobné zadání již existuje odpověď ve formě ano nebo ne. Podobné algoritmy slouží k nalezení k téměř optimálního řešení těžkých problémů [1].

2.1.3 NP optimalizační problémy

NP optimalizační problém je kombinačně optimalizační problém, který přidává následující podmínky [1].

- Velikost každého dosažitelného řešení je polynomiálně ohraničen na velikost dané instance x
- Všechny jazyky v I musí být spočítány v polynomiálním čase
- m musí být spočítatelná v polynomiálním čase

To jasně naznačuje, že odpovídající rozhodující problém je v NP. V počítačových vědách existuje spoustu zajímavých optimalizačních problémů a většina z nich má výše popsané vlastnosti a proto jsou označovány jako NP optimalizační problémy. Speciální případy, kdy pro problém existuje řešení zjistitelné v polynomiálním čase, se označuje jako P optimalizační problém.

Pro řešení tohoto typu úloh se využívá nepřeberné množství algoritmů, ale v mém případě využiji evolučního algoritmu. Konkrétně se jedná o diskrétní verzi SOMA, který je popsán níže a řeší plánovací úlohy.

3. Plánování

Plánování je proces sjednávání, řízení, optimalizace práce a pracovní zátěže ve výrobním procesu. Plánování se používá k plánování lidských zdrojů, plánování procesu produkce, získávání materiálu potřebného pro práci a zejména k efektivnímu využití výrobních prostředků a lidských zdrojů [2]. Jedná se o důležitý nástroj v praxi pro výrobní proces a strojírenství, kde má obrovský dopad na produktivitu výrobního procesu. Účelem plánování ve výrobním procesu je minimalizovat cenu samotné výroby a zajistit co nejkratší čas produkce výrobku. To je zajištěno tím, že výroba přesně ví, co kdy vyrábět, kdo to má vyrábět a na jakém zařízení proběhne výroba. Což jsou výstupy samotného plánovacího procesu pro zefektivnění výrobního procesu.

Výhody při využití plánování

- Vyrovnání zatížení pracovního procesu
- Přesné data dodávek výrobků
- Zvýšení efektivity výroby
- Snížení nákladů
- Sníženi změn ve výrobě

Existuje velké množství algoritmů, které řeší plánovací úlohy. V rámci této práce nás budou zajímat evoluční algoritmy, které dokáží úspěšně vyřešit tento problém. Řešení nemusí být nejlepší, ale budou se k němu přibližovat. Konkrétně se bude jednat o diskrétní verzi algoritmu SOMA.

3.1 Job Shop

Job Shop problém je optimalizační problém, který se vyskytuje v počítačových vědách a ve výzkumných odvětvích, ve kterých jsou spuštěny ideální úkoly přiřazeny k prostředkům v určitý čas. Snažíme se nalézt takovou posloupnost vykonávání úkolu, která bude nejvýhodnější. Tím je většinou myšleno vykonání v co nejkratším čase.

3.1.1 Základní verze Job Shop problému

Je dáno n úkolů $J_1, J_2, \dots, J_n$ různé velikosti, které musí být naplánovány na m počtu identických strojích, zatím co se snaží o minimalizaci času celkového běhu procesu [4]. To je, kdy doběhnou všechny úkoly na všech strojích.

Job Shop může mít různé variace a specifikace problémů [4]. Ty nejčastější, které se využívají jsou:

- Stroje mohou být nezávislé, navzájem spojené nebo jsou si rovny
- Stroje mohou vyžadovat určitou mezeru mezi úkoly nebo musí být spuštěny postupně bez jakékoliv prodlevy
- Cílem funkce může být minimalizace času běhu, maximalizace zpoždění, nedochvilnost úkolů a mnoho jiného
- Jednotlivé úkoly mohou mít nějaká kritéria, jako například úkol i musí být spuštěn před úkolem j
- Úkoly a stroje mají vzájemné vazby, jako například určitý úkol lze spustit pouze na určitém stroji
- Deterministická doba zpracování nebo pravděpodobnostní doba zpracování

3.2 Flow Shop

Jedná se o speciální případ Job Shop plánování, který si popíšeme podrobněji. Plánování Flow Shop problémů je třída plánovacích problémů, ve kterých řízení toku bude umožňovat vhodné sekvenční zpracování pro každý úkol na určité množině strojů [4]. Žádoucím faktorem je udržení kontinuálního toku zpracovávané úlohy s minimální dobou nečinnosti strojů a s minimální dobou čekání mezi úkoly. Plánování Flow Shop je speciální případ Job Shop plánování, ve kterém je striktně dané pořadí úloh, jak se budou vykonávat na jednotlivých strojích. Plánování Flow Shop se dá využít jak ve výrobních procesech, tak i v návrzích počítačových programů.

3.2.1 Formální definice

Nechť máme n strojů a m úkolů. Každý úkol obsahuje přesně n operací. i-tá operace úkolu musí být spuštěna na i-tém stroji. Stroj může vykonávat v jeden čas pouze jednu operaci. Pro každou operaci všech úkolů je přesně specifikován čas spuštění.

Operace v rámci jednoho úkolu musí být vykonány v určeném pořadí. První operace se spustí na prvním stroji. Potom, co se první operace dokončí, může být operace spuštěna na stroji druhém a tak dále, dokud nedojdeme k n-té operaci. Úkoly mohou být spuštěny v libovolném pořadí, každopádně definice problému nařizuje, aby bylo pořadí stejné na všech strojích. Snahou je nalézt takové uspořádání úloh, aby bylo dosaženo co nejkratší doby provádění úloh.

Není znám obecný algoritmus, který by nalezl řešení nejlepšího uspořádání úkolu. Proto se v této problematice hojně využívají evoluční algoritmy, které se za pomocí heuristiky snaží dopátrat k nejlepšímu výsledku.

3.2.2 Ukázkový příklad

Na obrázku níže je klasické zadání Flow Shop problému. Na 4 strojích(machines) provádíme 5 úkolů(jobs). V tabulce je přesně řečeno, jak dlouho trvá zpracování jednotlivých úkolů. Našim úkolem je nalézt takové pořadí provádění úkolů, aby byl zajištěna co nejmenší doba běhu (C_{max}). My si však ukážeme pouze, jak vypočítat dobu běhu problému, když bude pořadí úkolů (J_1, J_2, J_3, J_4, J_5). Kompletní řešení si ukážeme až v praktické části pomoci algoritmu DSOMA, protože jak jsem uvedl výše, není znám algoritmus, která by tento problém vyřešil přesně pro jakýkoliv vstup.

Tabulka 1 : Klasické zadání plánovací úlohy (zdroj: http://www.columbia.edu/~cs2035/courses/ieor4405.S16/perm.pdf)

úkoly	J_1	J_2	J_3	J_4	J_5
p_1, j_i	5	5	3	6	3
p_2, j_i	4	4	2	4	4
p_3, j_i	4	4	3	4	1
p_4, j_i	3	6	3	2	5

4 stroje, 5 úkolů

Algoritmus pro výpočet C_{max} problému Flow Shop je následující. Pro účely využijeme pomocnou tabulku. Výpočet začíná vždy u prvního stroje (p_1) a prvního úkolu (J_1). Doba prvního úkolu trvá 5. Doba trvání stejného úkolu, ale na stroji p_2 už bude součet trvání na prvním a druhém stroji. To znamená $5 + 4 = 9$ a zapíšeme do tabulky. Tento postup se opakuje pro první úkol, dokud nenarazíme na poslední stroj. Pro další úkol J_2 se již výpočet mírně komplikuje. Na prvním stroji stačí ještě pouze sečíst dobu trvání prvního a druhého úkolu, což je v našem případě $5 + 5 = 10$. Pro druhý stroj však už musíme využít přepočtového vzorce níže [4].

$$C_{m,j} = \max\left(C_{i-1,j}, C_{i,j-1}\right) + P_{i,j} \tag{3.1}$$

Je zde funkce $max()$, ta určuje, která doba trvání je zatím vyšší. Jestli doba stejného úkolu u stroje o index nižší nebo doba na stejném stroji, ale u úkolu o index nižší. Zvolí se hodnota vyšší a k té se připočítá doba trvání úkolu. U našeho příkladu je to takto $max(10, 9) + 4 = 14$. Ale takto se provede algoritmus pro celý zadaný příklad.

Tabulka 2: Pomocná tabulka Flow Shop

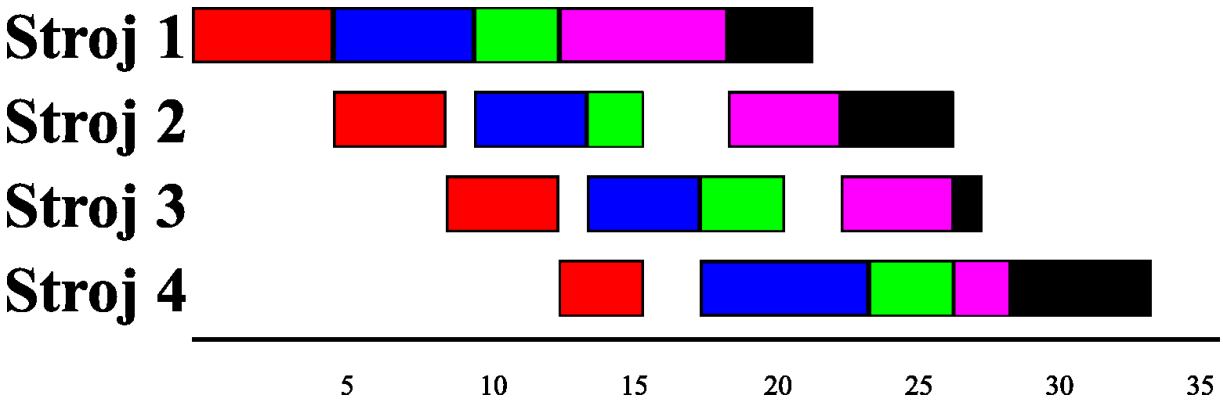
úkoly	J_1	J_2	J_3	J_4	J_5
p1	5	10	13	19	22
p2	9	14	16	23	27
p3	13	18	21	27	28
p4	16	24	27	29	34

Následuje sumární výpočet C_{max}, který se provede podle vzorce

$$\Im = \sum_{j=1}^{n} C_{m,j}\,, \tag{3.2}$$

kde, sumárně sčítáme dobu provedení všech úkolů na všech strojích $C_{m,j}$. Našemu ukázkovému příkladu vyšlo $C_{max} = 34$.

Výsledek se dá prezentovat i grafickou formou a na dalším obrázku je graf zadané úlohy. Stejná barva označuje úkol, který nemůže byt zpracováván v jeden okamžik na více strojích. V grafu se proto vyskytují mezery, kdy se na stroji pouze čeká, než se provede úkol na předešlém stroji. Tak jak můžete vidět na obrázku níže.



Obrázek 1: Jedno z řešení plánovací úlohy (zdroj: http://www.columbia.edu/~cs2035/courses/ieor4405.S16/perm.pdf)

4. Heuristický algoritmus

Jedná se o algoritmy, které pro své výpočty využívají heuristiku. Heuristika využívá zkusmého řešení problému, pro které neexistuje algoritmus, který by problém řešil. Výsledkem heuristického algoritmu je nejčastěji pouze přibližné řešení, které je založeno na náhodě, odhadu, intuici. Zpočátku taková řešení mohou být nepřesná, ale časem se mohou zlepšovat, i když nikdy nemusíme dosáhnou přesného řešení. Využívá se zejména z důvodu jednoduché implementace a rychlosti výpočtu. Typicky tyto algoritmy obsahují možnost pro další pokračování výpočtu. Bohužel heuristický algoritmus může selhat. To znamená, že řešený problém nevydá správné řešení, nevypočítá žádné řešení nebo se algoritmus ani nezastaví a běží dále. Pro jeden problém může existovat více heuristických přístupů. Mohou být vhodné nebo přímo navržené pro daný problém anebo se specializují na rychlost výpočtu nebo kvalitu výsledku. Výběr pokračování řešení může být heuristický nebo náhodný. Nejčastěji se však využívá kombinace obou přístupů a využíváme heuristiky s využitím malé pravděpodobnostní náhody. Takto kombinovat můžeme i použité heuristiky v rámci jednoho problému. Mezi algoritmy využívající přístupu heuristiky patří [8].

- Evoluční a genetický algoritmus – Algoritmus využívající principu přirozené selekce. Úspěšně se využívá pro řešení počítačových heuristických problémů a dokáže nalézt velice kvalitní řešení i u složitých problémů. Využívá se pro řešení mnoha technických a matematických problémů.
- Lokální hledání – Nejjednodušší metoda řešení problémů. Využívá se zejména pro řešení matematických problémů.
- Iterativní metoda – Postupně nachází lepší řešení a zužuje tak oblast řešení.

Heuristické algoritmy se využívají zejména u optimalizačních problémů, kde se hledá extrém, ať se jedná o maximum či minimum. Využívá se i u rozhodovacích problémů, kde je odpovědí ANO/NE. Potom se mluví o pravděpodobnostním algoritmu.

5. Evoluční algoritmus

Evoluční algoritmus (EA) je heuristický postup, který se snaží za pomoci využití principů evoluční biologie nalézt řešení nějakého složitého problému, pro který neexistuje výpočetně vystopovatelný algoritmus, který by jej přesně řešil. Tyto algoritmy využívají techniky napodobující evoluční procesy, známe z biologie. Konkrétně se jedná o dědičnost, mutace, přirozený výběr a křížení. Tyto techniky mají za úkol vylepšit populaci tím, že vylepšují či zušlechťují řešení zadaného problému a tím získáváme postupně lepší řešení problému. Pod evoluční algoritmy patří Genetické algoritmy, Evoluční strategie, Evoluční programování [9].

Principem evolučního algoritmu je postupná tvorba nových generací různých řešení problémů, které se snažíme vyřešit pomocí vyložení Darwinovy teorie či jiné teorie kterou můžeme nalézt v biologii. Pro účely řešení evolučních algoritmu se využívá populace, kde každý jedinec představuje právě jedno řešení daného problému. Tím, jak probíhá evoluce, se řešení postupně zlepšují. Je mnoho způsobů, jak se reprezentuje řešení. Pro tento algoritmus je typické, že prvotní populace obsahuje jedince vytvořené úplně náhodně. Při přechodu na novou generaci se pro každého jedince spočítá fitness funkce, která vyjadřuje, o jak kvalitního jedince v populaci se jedná. Pomocí této kvality jsou stochasticky vybráni jedinci, kteří mohou být modifikováni. Takto nakonec vznikne nová a vylepšená populace. Tyto algoritmy mají ukončovací podmínku a ta je buď nalezení dostatečně kvalitního řešení problému, ukončeni po předem dané době nebo ukončení po vytvoření určitého počtu generací jedinců.

Evoluční algoritmus má toto obecné schéma, které je zpravidla totožné pro všechny podkategorie podobných problémů [9].

1. Inicializace – Vytvoření první populace, zpravidla náhodně vygenerovanými jedinci.
2. Začátek cyklu – Pomocí určité výběrové metody vybereme z populace nejkvalitnější jedince.
3. Využití evolučního procesu – Z vybraných jedinců vygeneruje nové jedince za pomoci evolučních procesů, čímž vznikne nová a pokud možno lepší populace.
    i. Evoluce – Manipulace s populaci podle určitých evolučních pravidel.
4. Výpočet fitness – Výpočet kvality jednotlivých jedinců.
5. Konec cyklu – Pokud není splněna zastavovací podmínka, pokračujeme dále na začátek cyklu.
6. Konec algoritmu – Nejlepší jedinec na konci algoritmu představuje nejlepší řešení našeho problému.

5.1 Samo-Organizující Migrační Algoritmus

Tato část kapitola velice stručně vysvětlí princip Samo-Organizujícího Migračního Algoritmus. V druhé částí kapitoly podrobně popíšu diskrétní verzi tohoto algoritmu, kterou budu potřebovat pro implementaci algoritmu v praktické části práce. Podrobně popíši každou funkci a pro názornost uvedu i pseudokódy jednotlivých části kódu.

5.1.1 Samo-Organizující Migrační Algoritmus (SOMA)

SOMA je algoritmus, který existuje již od roku 1999 a pracuje na principu vektorových operací. Jedná se o zástupce evolučního algoritmu, který provádí výpočty nad jedinci uspořádanými v populaci. Je však odlišný od ostatních evolučních algoritmů, protože nevytváří nové jedince, ale jen je posouvá v prostoru. Primární určení algoritmu je pro nalezení globálního extrému spojité funkce. Jako ostatní evoluční algoritmy i SOMA algoritmus obsahuje parametry, které je potřeba vhodně nastavit. Jsou to $PathLengh$ (označuje délku cesty), $Step$ (délka kroku), PRT (velikost perturbačního vektoru), D (dimenze problému), $PopSize$ (velikost populace) a $Migrace$, která určuje počet migračních cyklů [6].

V algoritmu SOMA se označení mutace přejmenovalo na perturbace z důvodu lepšího popisu. Pohyb jedince prostorem řešení je náhodně rušen, tedy je perturbován. Množství změněných parametrů jedince určuje parametr PRT. Za použití tohoto parametru se vytvoří $PRTVector$, který se generuje pro všechny jedince zvlášť a platí pouze jeden migrační cyklus pro jediného jedince populace. Pohyb jedinců po prostoru probíhá pomocí diskrétních skoků tak, že se jedince v prostoru pohybuje ke zvolenému vůdci populace. Postupně se ověřují jednotlivé skoky a pokud je výsledek lepší, nahradí jedince z populace. Potom může být provedeno další migrační kolo, které muže vést k dalšímu vylepšení řešení.

5.1.2 Diskrétní Samo-Organizující Migrační Algoritmus (DSOMA)

Diskrétní Samo-organizační Migrační Algoritmus (DSOMA) je diskrétní varianta algoritmu SOMA, která je vyvinuta k řešení kombinatoricky optimalizačních problémům na bázi permutace. Stejná ideologie vzorkování prostoru mezi dvěma jedinci zůstala. Úkolem DSOMA je prohledat prostor mezi dvěma jedinci tak, že zmapuje diskrétní prostor mezi nimi [7].

Hlavním přínosem tohoto algoritmu je vzorkování skokových sekvencí mezi jedinci v populaci a procedura pro vytváření pokusných jedinců z těchto navzorkovaných elementů skokových sekvencí.

5.1.2.1 Přehled algoritmu DSOMA

I. Inicializační fáze

1. Generování populace: Vytvoření počáteční populace.
2. Ohodnocení fitness: Každý jedinec je ohodnocen funkci vhodnosti - fitness.

II. DSOMA

1. Vytvoření sekvence skoků: Výběr dvou jedinců. Počet možných pozic skoku je vypočítán mezi odpovídajícími elementy jedinců.
2. Zkonstruování pokusných jedinců: Využití všech pozic skoku. Vygeneruje se určitý počet pokusných jedinců, kde každý element jedince je vybrán z porovnání mezi dvěma jedinci.
3. Oprava pokusných jedinců: Všichni pokusní jedinci jsou zkontrolováni, jestli jsou korektní. Nekompletní a chybní jedinci se opraví.

III. Výběr

1. Výběr nových jedinců: Noví jedinci jsou znovu ohodnoceni jejich fitness a jedinci s lepší fitness nahradí staré jedince s fitness horší.

IV. Generace

2. Iterace: Opakování algoritmu

DSOMA vyžaduje parametry, které jsou uvedeny v tabulce níže. Hlavní změnou v diskrétní verzi SOMA je parametr J_{min}, který určuje minimální počet skoků mezi dvěma jedinci. Parametry $PathLength$, $StepSize$ a $PRTVector$ úplně chybí, protože jsou dynamicky vypočteny algoritmem DSOMA.

Tabulka 3: Parametry DSOMA (zdroj: Studijní materiály BIC – VŠB-TUO Ostrava)

Název	Rozsah	Typ	Popis
J_{min}	(1+)	Kontrolní	Minimální počet skoků
Populace	10+	Kontrolní	Počet jedinců v populaci
Migrace	10+	Ukončovací	Počet iterací

5.1.2.2 Inicializační fáze

Populace je inicializována jako permutační plán, kde jednotliví jedinci jsou velcí podle velikosti řešeného problému. Protože se zde jedná o inicializaci původní populace, je potřeba vyplnit elementy jedinců náhodnými čísly. Funkce generátoru náhodných čísel $rand()$ vrátí hodnotu mezi 0 a 1. Funkce $INT()$ zaokrouhlí náhodné číslo na nejbližší celočíselný výsledek. Je potřeba zajistit, aby se vygenerovaná čísla v rámci jedince neopakovala. To je zajištěno podmínkou if (pokud). Přehledně je algoritmus generování populace popsán v rovnici 5.1 [7].

$$x_{i,j}^{0} = \begin{cases} 1 + INT\,(rand() * (N-1)) \\ \text{Pokud } x_{i,j}^{0} \notin \{x_{i,j}^{0}, \dots, x_{i,j-1}^{0}\} \end{cases} \tag{5.1}$$

$$i = 1, \dots, \beta; j = 1, \dots, N$$

Každý jedinec je ohodnocen funkcí fitness, která určí jeho vhodnost, a nejlepší jedinec v populaci bude označen jako L (leader - vůdce). Tento index určí vůdce jako X_L^O s jeho vhodnosti C_L^O. Funkce vhodnosti se použije podle problému, který právě řešíme. Obecně se uvádí jako na rovnici níže [7].

$$C_i^0 = \Im\left(X_i^0\right), \quad i = 1, \dots, \beta \tag{5.2}$$

Po vygenerování počáteční populace je migrační počítadlo t nastaveno na 1 kde $t = 1, \dots, M$ a index jedince i, kde $i = 1, \dots, \beta$.

Kompletní pseudokód generování populace je uveden níže [7].

1. Pro $i = 1, 2, \dots, \beta$ dělej následující:
    a. Pro $j = 1, 2, \dots, N$ dělej následující:
        i. Náhodně generuj hodnotu $x_j = rnd\ int\ [1, N]$
        ii. Dokud $x_j \notin X_i$
            Náhodně generuj hodnotu $x_j = rnd\ int\ [1, N]$
        iii. Vlož $x_j \rightarrow X_i$
    b. Vlož $X_i \rightarrow P$
    c. Vypočti fitness pro X_i jako $C_i = \Im(X_i)$
2. Nastav $C_L = \min(C)$
3. Nastav nejlepší index $b = \text{index}\min(C)$
4. Nastav $X_L = X_b$

Pseudokód 1: Pseudokód pro generování populace (zdroj: [7])

5.1.2.3 Vytvoření sekvencí skoků

DSOMA funguje tak, že vypočítá počet diskrétních skoků, o které se jedinec posune. DSOMA obsahuje parametr J_{min}, který nahradil parametr $PathLength$ z klasického algoritmu SOMA. Parametr určuje minimální počet skoků mezi dvěma jedinci.

Z populace vezmeme dva jedince. Jeden bude vůdce (X_L^t) a druhý bude přilehlý jedinec (X_i^t). Celkový počet všech možných skoků jedinců je označen jako J_{max} . Parametr J_{max} popisuje rozdíl mezi přilehlými hodnotami elementů jedince. Vektor J, který má velikost N ukládá rozdíl mezi přilehlými elementy v jedincích. Funkce $mode()$ získá nejběžnější číslo ve vektoru J a označí ji jako J_{max}. Přesné rovnice jsou uvedeny dále v 5.3 [7].

$$J_j = \left|X_{i,j}^{t-1} - X_{L,j}^{t-1}\right|, j = 1, \dots, N$$

$$J_{max} = \begin{cases} mode(J) & \text{Pokud } mode(J) > 0 \\ 1 & \text{Jinak} \end{cases} \tag{5.3}$$

Délka kroku (s) se vypočítá jako celočíselný podíl mezi vyžadovanými skoky J_{max} a možnými skoky J_{min}. Pokud však J_{min} je větší než J_{max} , délka kroku se nastaví na 1. To je zobrazeno v rovnici 5.4 [7].

$$s = \begin{cases} \left\lfloor \frac{J_{max}}{J_{min}} \right\rfloor & \text{Pokud } J_{max} \geq J_{min} \\ 1 & \text{Jinak} \end{cases} \tag{5.4}$$

Vytvoření skokové matice G, která obsahuje všechny možné pozice skoku, které byly vypočteny pomocí rovnice 5.5. Porovnávají se jednotliví jedinci s vůdcem populace a podle podmínky se rozhodne, o kolik se jedinec k vůdci posune [7].

$$\mathrm{G_{l,j}} = \begin{cases} x_{i,j}^{t-1} + s * l & \text{Pokud } x_{i,j}^{t-1} + s * l < x_{L,j}^{t-1} \; a \; x_{i,j}^{t-1} < x_{L,j}^{t-1} \\ x_{i,j}^{t-1} - s * l & \text{Pokud } x_{i,j}^{t-1} + s * l < x_{L,j}^{t-1} \; a \; x_{i,j}^{t-1} > x_{L,j}^{t-1} \\ 0 & \text{Jinak} \end{cases} \tag{5.5}$$

$$j = 1, \dots, N; l = 1, \dots, J_{min}$$

Kompletní pseudokód vytvoření sekvencí skoků vypadá následovně a je podrobně popsán dále [7].

1. Pro $i = 1,2, \dots, \beta$ dělej následující:
    - *a.* Pro $j = 1,2, \dots, N$ dělej následující:
        - *i.* $J_j = \left| X_{L,j} - X_{i,j} \right|$
    - *b.* $J_{max} = mode(J)$
    - *c.* Pokud $J_{max} \geq J_{min}$

      $s = \left\lfloor \frac{J_{max}}{J_{min}} \right\rfloor$

      Jinak $s = 1$
    - *d.* Pro $j = 1,2, \dots, N$ dělej následující:
        - *i.* Pro $l = 1,2, \dots, J_{min}$
            - *A.* Pokud $x_{i,j} < x_{L,j}$

              $G_{l,j} = x_{i,j} + s * l$

            Ale pokud $x_{i,j} > x_{L,j}$

              $G_{l,j} = x_{i,j} - s * l$

            Jinak $G_{l,j} = 0$

Pseudokód 2: Pseudokód vytváření sekvencí skoku (zdroj: [7])

5.1.2.4 Zkonstruování pokusných jedinců

Pro každou skokovou sekvenci dvou jedinců je vytvořeno celkem J_{min} nových jedinců, kteří jsou vypočteni ze skokových pozic. Vytvoříme novou dočasnou populaci $H(H = \{Y1, \dots, Y_{J_{min}}\})$, ve které každý nový jedinec $Y_w(w = 1, \dots, J_{min})$ je postupně vytvořen z matice G. Všechny elementy jedinců $Y_{w,j}$ $(Y_w = \{y_{w,j}, \dots, y_{w,N}\}, j = 1, \dots, N)$ indexují svoji hodnotu z odpovídajícího j-tého sloupce v matici G. Každá l-tá pozice určitého elementu jedince se sekvenčně kontroluje v $G_{l,j}$, jestli již prvek existuje v jedinci Y_w. Pokud se jedná o nový element, jedinec jej akceptuje a odpovídající l-tá hodnota se nastaví na nula. $G_{l,j} = 0$ [7].

Procedura je souhrnně na pseudokódu níže, kde je popsán algoritmus vytváření nových jedinců [7].

1. Pro $w = 1,2, \dots, J_{min}$ dělej následující:
    a. Pro $j = 1,2, \dots, N$ dělej následující:
        i. Pro $l = 1,2, \dots, J_{min}$ dělej následující:
            A. Pokud$(G_{l,j} \notin \{y_{w,1}, \dots, y_{w,y-1}\}\, A\, G_{l,j} \neq 0)$

$$y_{w,j} = G_{l,j}$$

$$G_{l,j} = 0$$

        ii. Pokud $y_{w,j} == NULL$

$$y_{w,j} = 0$$

Pseudokód 3: Pseudokód zkonstruování nových jedinců (zdroj: [7])

5.1.2.5 Oprava pokusných jedinců

Někteří nově vytvoření pokusní jedinci nemusí obsahovat korektní permutační plán. Skokoví jedinci $Y_w(w = 1, \dots, J_{min})$, jsou vytvořeni tak, že každý nemožný element jedince $y_{w,j}$, je označen nulou. Postupně vezmeme každého jedince Y_w z H a provedeme následující proceduru.

Máme množiny A a B, kde množina A je inicializovaný permutační plán $A = \{1, \dots, N\}\, a\, B$ je doplněk jedince Y_w k množině A. Tak jak je uvedeno v rovnici 5.6 [7].

$$B = A \backslash Y_w \tag{5.6}$$

Pokud po této operaci pro jedince vychází prázdná množina bez jakéhokoliv prvku $B = \{\}$, potom je jedinec platný a obsahuje korektní permutační plán. V tomto případě už tedy není potřeba oprava jedince.

Každopádně, pokud pro jiného jedince množina B obsahuje nějaké prvky, pak tyto hodnoty jsou právě ony chybějící elementy v jedinci Y_w. Bude potřeba provést opravnou proceduru. Prvním krokem je náhodně promíchat hodnoty v množině B. Poté postupně procházet elementy $Y_{w,j}$ $(j = 1, \dots, N)$ v jedinci Y_w a každou pozici, kde je element $y_{w,j}$ roven nule, nahradíme hodnotou z množiny B. B_{size} je celkový počet prvků, které jsou v množině B. Pseudokód opravné procedury můžete vidět na pseudokódu dále [7].

1. Pro $w = 1,2, \dots, J_{min}$ dělej následující:
    a. $B = A \setminus Y_w$
    b. Pokud $B \neq \{\}$

        Náhodně proházej elementy v B.

        i. Pro $j = 1,2, \dots, N$ dělej následující:

            A. Nastav index $h = 1$

            B. Pokud $y_{w,j} == 0$

                $y_{w,j} = B_h$

                $h = h + 1$

    c. Ohodnoť fitness u pokusných jedinců jako:

        $\gamma_w = \Im(Y_w)$

    d. Pokud $w == 1$

        $Y_{best} = Y_w$

    e. Pokud jinak $\gamma_w < Y_{best}$

        $Y_{best} = Y_w$

Pseudokód 4: Pseudokód opravy pokusných jedinců (zdroj: [7])

Po opravě všech jedinců v H proběhne jejich ohodnocení účelovou funkcí. Ta se uloží do pole velikosti J_{min}, tak jak je uvedeno na rovnici 5.7.

$$\gamma_w = \Im(Y_w), \quad w = 1, \dots, J_{min} \tag{5.7}$$

5.1.2.6 Aktualizace populace

Algoritmus 2 Opt lokální hledání se aplikuje na nejlepšího pokusného jedince označeného Y_{best}, kterým je jedinec s minimální hodnotou účelové funkce. Po použití lokálního vyhledávání porovnáme nového jedince s jedincem doposud nejlepším X_i^{t-1}. A to tak, že porovnáme jejich hodnoty účelových funkcí. Pokud je nový jedinec lepší, tedy hodnota účelové funkce je menší, nahradíme starého jedince tímto novým. Na rovnici 5.8 vidíme podrobně rovnici s podmínkou pro porovnání jedinců [7].

$$X_i^t = \begin{cases} Y_{best} & \text{Pokud } \Im(Y_{best}) < C_i^{t-1} \\ X_i^{t-1} & \text{Jinak} \end{cases} \tag{5.8}$$

Pokud nově vzniklý jedinec je natolik dobrý, že je nejlepším z celé populace, tak se nastaví jako nejlepší jedinec populace. To je ukázáno na rovnici 5.9 [7].

$$X_{best}^t = \begin{cases} Y_{best} & \text{Pokud } \Im(Y_{best}) < C_{best}^t \\ X_{best}^{t-1} & \text{Jinak} \end{cases} \tag{5.9}$$

5.1.2.7 Iterace

Sekvenčně inkrementujeme populační počítadlo i o jedna. Další jedinec X_{i+1}^{t-1} je vybrán z populace a začne provádět vzorkování k vůdci, který je označen jako X_L^{t-1}. Je důležité zmínit, že v rámci jedné migrace se nemění vůdce populace [7].

5.1.2.8 Migrace

Poté, co všichni jedinci vykonali pokusy se přiblížit k vůdci populace, migrační počítadlo se inkrementuje o jedničku. Iterátor jedince i se nastaví znovu na jedničku. To je tak jako na začátku populace. A celá smyčka se provede znovu od bodu Vytváření skokových sekvencí po Iteraci [7].

5.1.2.9 2 Opt lokální vyhledávání

Pro lokální vyhledávání je v DSOMA využitý 2 Opt lokální vyhledávací algoritmus (2 Opt local search algorithm). Důvodem, proč je zde využit 2 Opt lokální vyhledávání, je to, že se jedná o nejjednodušší variantu v k-Opt třídě problémů. DSOMA provádí vzorkování mezi dvěma jedinci v k-dimenzionálním prostoru a lokální vyhledávání zlepšuje jedince. Tato metoda zvyšuje šance k nalezení nového vůdce po každé skokové sekvenci [7].

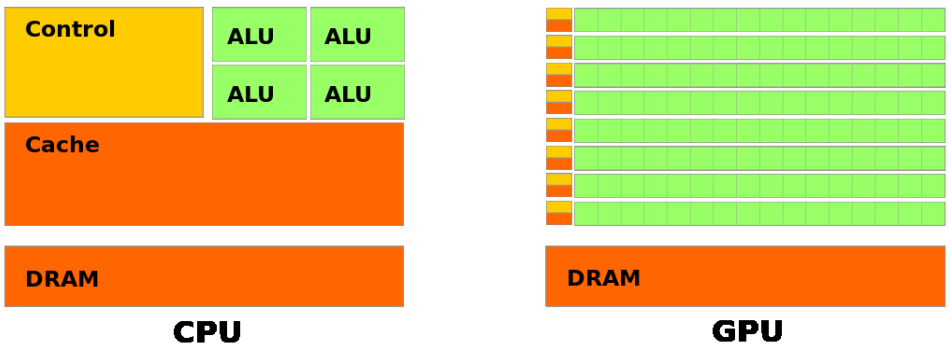
Složitost lokálního vyhledávání je $O(n^2)$. To představuje většinu časové složitosti celého algoritmu DSOMA. Celková časová složitost DSOMA pro jednu migraci je $O(n^3)$.

6. CUDA

CUDA (Compute Unified Device Architecture) je hardwarová a softwarová architektura, která umožňuje na grafické kartě (GPU) spouštět programy napsané nejčastěji v jazyce C/C++. Lze však využít i nepřeberné množství dalších programovacích jazyků jako jsou FORTRAN, PyCUDA, OpenCL a mnoho dalších. Tuto architekturu vyvinula společnost NVIDIA a je dostupná pouze na grafických kartách této společnosti. Společnost NVIDIA je v současnosti jeden z největších výrobců grafických karet a vede nekonečný konkurenční boj se společností AMD (dříve známé jako Ati). To je jeden z hlavních důvodů, proč se CUDA nerozšířil až tak rychle globálně do světa. Dalším z důvodu je kompatibility pouze s vlastními grafickými kartami. Obě společnosti jsou členy Khronos Group, která vyvíjí OpenCL, což je alternativní architektura, která běží na grafických kartách obou zmíněných společností. Není však ani zdaleka tak rozšířená jako CUDA [5].

Technologii CUDA představila společnost NVIDIA v roce 2006. Do té doby byly výpočetní jádra na grafické kartě schopny zpracovávat pouze grafický obsah. Nyní se však grafická jádra unifikovala a programátor mohl libovolně pracovávat s jádry jako u procesoru (CPU). Původně byla zamýšlená podpora pouze pro profesionální karty NVIDIA Tesla, což jsou karty speciálně zkonstruovány pro výpočty a neobsahují například žádný grafický výstup. To se však brzy změnilo a již v roce 2007 byla vypuštěna softwarová vývojová sada (SDK) pro stolní grafické karty společnosti NVIDIA. Jsou však podporovány pouze karty architektury G80 a novější. V té době se jednalo o grafické karty GeForce 8 série. Byl potřeba nejnovější grafický ovladač. V historii měla CUDA spoustu limitací a neobsahovala ani zdaleka kompletní sadu operací a funkcí, které byly potřeba pro bezproblémové implementování problémů. To však již v nejnovější SDK 7.5 není pravda a CUDA se stala konkurenceschopnou s programováním na CPU. Dokonce se nebojím tvrdit, že při řešení některých problémů je již daleko lepší, než klasické programování.

Největším přínosem technologie CUDA je právě paralelizace problému. V současné době nejvýkonnější grafické karty NVIDIA obsahují tisíce výpočetních jader a pokud dokážeme problém, který se snažíme vyřešit, rozdělit na problémy menší, které mohou běžet paralelně, tak doba výpočtu může být mnohokrát rychlejší, než při klasickém programování na CPU, kde máme pouze omezené množství výpočetních jader.



Obrázek 2: Blokové schéma GPU vs. CPU (zdroj: docs.nvidia.com)

Na obrázku výše můžete vidět blokové schéma grafické karty NVIDIA. Jednotlivá jádra jsou sdružována do větších bloků. Ty mají malou společnou cache paměť. Samozřejmě nesmí chybět i globální větší paměť. V porovnání s procesorem můžete vidět, že se jedno a velice rozdílné přístupy, kde procesor obsahuje omezený počet jader, ale ty jsou mnohem robustnější a výkonnější než u grafické karty [10].

Popis životního cyklu programu v CUDA vypadá následovně. Největší změna oproti klasickému přístupu programování na procesoru počítače je nutnost překopírovat data, nad kterými se bude pracovat do paměti grafické karty. To podle velikosti problému může byt ovlivňující faktor výkonu aplikace. Pro zobrazení výsledků výpočtu je nutno data překopírovat zpět do paměti počítače.

- Vytvoření dat v paměti RAM počítače
- Překopírováni do paměti RAM grafické karty
- Provedení výpočtů nad těmito daty
- Zkopírováni dat zpět do RAM počítače
- Interpretace výsledku

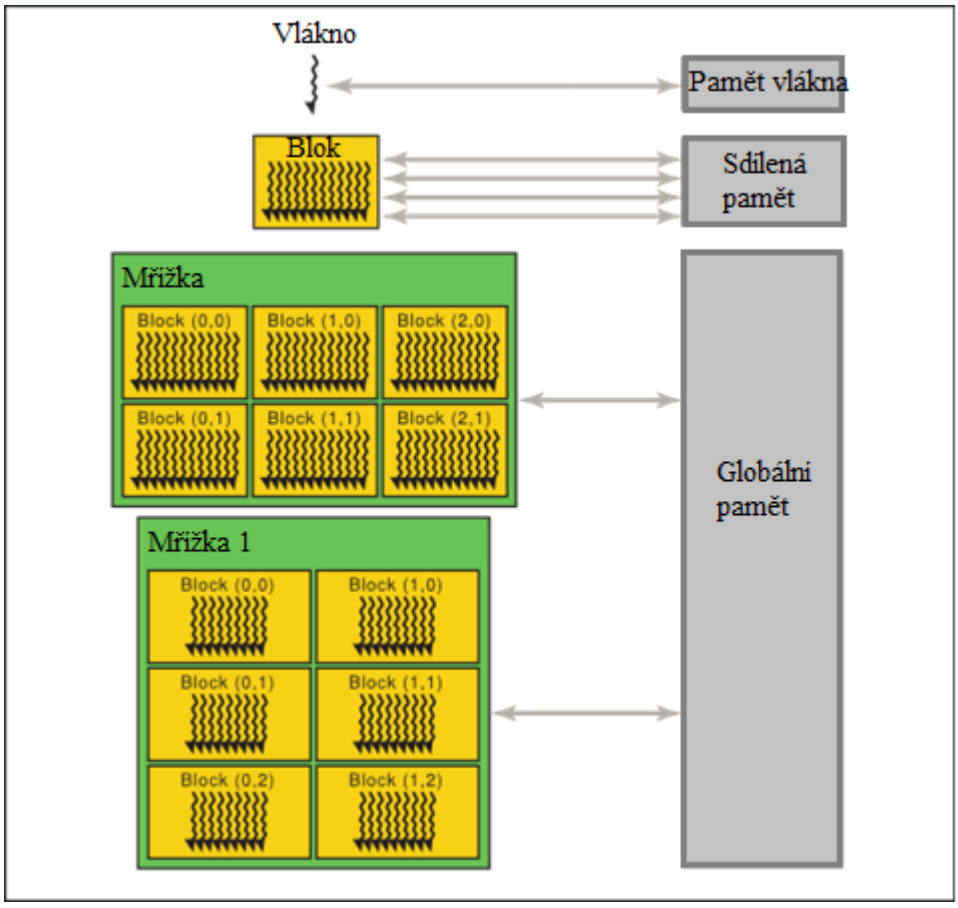
6.1 Softwarový model

6.1.1 Blok (thread block)

Vlákna jsou organizována do 1D, 2D nebo 3D bloků, které v rámci jednoho bloku mohou sdílet data a jejich běh je synchronizován. Počet vláken na blok je závislý na výpočetních možnostech grafické karty. Každé vlákno v bloku je identifikovatelné unikátním indexem přístupným ve spuštěném kernelu přes vestavěnou proměnnou threadIdx [10].

6.1.2 Mřížka

Bloky jsou organizovány do 1D, 2D a 3D mřížek. Blok v rámci mřížky dokážeme identifikovat za pomoci unikátního indexu přístupného ve spuštěném kernelu přes zabudovanou proměnou blockIdx. Každý blok vláken pracuje nezávisle na ostatních, aby byla možná škálovatelnost výpočtu [10].



Obrázek 3: Softwarový model CUDA(zdroj: docs.nvidia.com)

Obrázek 3 názorně zobrazuje, jak jednotlivá vlákna(Thread) mohou přistupovat pouze ke své paměti. Blok(Block) vláken však již má sdílenou paměť pomocí niž si může v rámci jednoho bloku vyměňovat data. Mřížky(Grid) mezi sebou mohou komunikovat pouze pomocí globální paměti umístěné na grafické kartě [10].

6.2 Paměťový model

Grafická karta obsahuje 6 různých druhů paměti, se kterými může programátor pracovat [10].

- **Pole registrů** - Je umístěno na jednotlivých jádrech grafické karty. Každé vlákno přistupuje pouze ke svým registrům.
- **Lokální paměť** - V případě, že se vyčerpá místo pro registry tak se využije lokální paměť. Tato paměť je přístupná pouze jedinému vláknu, ale fyzicky je umístěná v globální paměti. Proto je přístup pomalejší než například k sdílené paměti.
- **Sdílená paměť** - Po registru je to jediná paměť, která je umístěna přímo v jádru. Přistupují k ní všechna vlákna v rámci jednoho bloku. Jedná se o nejefektivnější paměť, pokud potřebujeme přistupovat ke stejným datům v rámci bloku.

- **Globální paměť** - Tato paměť je sdílená všemi jádry a vůbec se neukládá do cache paměti, proto je přístup do ní relativně pomalý. Ukládají se zde hlavně konečné výsledky programu.
- **Paměť konstant** - Slouží pouze pro čtení a stejně jako globální paměť je sdílená se všemi jádry. Jediný rozdíl je v tom, že je pro ní v čipu vyhrazeno místo v L1 cache. Výsledky rozesílá broadcastem.
- **Paměť textur** - Je sdílená mezi jádry a určená pouze pro čtení a navíc disponuje cache pamětí. Je optimalizována pro 2D prostorovou lokalitu, proto vlákna ve stejném warpu, které čtou z blízkých texturovacích souřadnic, dosahují lepších výkonů. Pro účely programování není využívaná.

7. Praktická implementace v CUDA

V této kapitole podrobně popíšu, jak jsem naimplementoval jednotlivé části DSOMA algoritmu. Postupně odůvodním, proč jsem zvolil takový postup. Jednotlivé odstavce budou metody, tak jak je mám naimplementovány. V pozdější části kapitoly testování se zaměřím na efektivitu algoritmu a v tabulce znázorním, jak dlouho trvá výpočet jednotlivých částí kódu.

Pro účely této práce bylo potřeba nadefinovat několik konstant, které budu využívat při implementaci algoritmu DSOMA. Jsou dále hojně využívány v jednotlivých funkcích a procedurách.

- MIGRATIONS – určuje, kolik se provede migračních cyklů v rámci výpočtu
- DIMENSION – značí, jaké velikosti budou jednotliví jedinci
- POPSIZE – označuje počet jedinců v populaci
- Jmin – říká, kolik je povoleno skoků jedince
- LB – určuje minimální hodnotu jedince
- HB – značí maximální hodnotu jedince
- MACHINECOUNT – určuje, kolik strojů obsahuje zadání Flow Shop problému

Níže uvádím posloupnost provádění funkci algoritmu DSOMA. Jednotlivé kroky jsou na sobě závislé a není možno je jakkoliv přeházet nebo vykonávat paralélně. Vždy uvádím, jak se funkce jmenuje v mém kódu a vysvětluju, co funkce provádí.

- Generování populace – generatePopulation()
- Nalezení fitness u jedinců – findFitnessCostFlowshop()
- Nalezení vůdce – findLeader()
- **Začátek migračního cyklu**
- Vypočítej pole J – calculateJ()
- Vynulování Histogramu pro J – jHistogramReset()
- Vypočítaní histogramu pro J – jHistogram()
- Výpočet skoku s – calculatesFromJmax()
- Generování pole G – calculateG()
- Vytvoření pokusných jedinců – constructingTrial()
- Opravy jedinců – repairTrial(), repairTrial2()
- Nalezení fitness v H findFitnessCostFlowshop()
- Aktualizace populace – populationUpdate()
- **Konec migračního cyklu**
- Nalezení nejlepšího jedince – minFit()

Vstupní parametry aplikace je možno měnit v konfiguračním souboru parameters.txt.

7.1 Implementace funkcí

V odstavcích popíšu, jak jsem naimplementoval jednotlivé procedury algoritmu DSOMA v CUDA. V textu se budou vyskytovat pojmy s prefixem *d_* které označují, že tato proměnná či pole je umístěna v paměti grafické karty.

7.1.1 Generování populace (generatePopulation())

Jelikož je generování prvotní populace založené na generování náhodných permutací jednotlivých jedinců, bylo potřeba zajistit, aby jednotliví jedinci získali své elementy pomocí kvalitního míchacího algoritmu. To se mi podařilo pomocí Fisher-Yates algoritmu. Ten zajistil, že všechny výsledné permutace mají stejnou šanci na vygenerování.

7.1.1.1 Fisher-Yates míchací algoritmus

Jedná se o algoritmus, který generuje náhodné permutace konečné množiny. Jednodušeji řečeno promíchá množinu čísel. Algoritmus efektivně dává všechny elementy dohromady a postupně rozhoduje, který prvek bude vybrán, dokud v hromadě nezbyde žádný element. Algoritmus produkuje nestranné permutace. Lépe řečeno, všechny permutace mají stejnou šanci na vygenerování. Moderní verze tohoto algoritmu jsou efektivní a zabírají přiměřený čas tomu, jak velkou množinu čísel promícháváme.

Pro implementaci jsem si zvolil inside-out (opačnou) verzi Fisher-Yates algoritmu, která je určena hlavně pro účely programovací.

```
//FISHER YEATS Shuffle
#pragma unroll
for (int i = 0; i < DIMENSION + 1; ++i) {
      int j = urandint(0, i, state);
      if (j != i)
            seq[i] = seq[j];
            seq[j] = vals(i);
      }
```

Výpis kódu 1: Implementace Fisher-Yates algoritmu

Na obrázku výše je hlavní část kódu, která zajišťuje samotné promíchání elementů jednoho jedince. Algoritmus funguje tak, že pro každý index jedince si vygeneruje pomocí metody *urandint()* náhodné celé číslo z intervalu 1 až dimenze jedince. Tato funkce je založená na generování náhodných čísel *curand()*, která jsou součástí knihovny CUDA. Porovnají se čísla *i* (index dimenze) s *j* (náhodné celé číslo) a pokud si nejsou rovny, uložíme do výsledného pole *seq* na index *i* obsah pole *seq* na indexu *j*. Následně do seq na index *j* vložíme číslo z vals index *i*, kde je uložená posloupnost celých čísel od 1 po dimenzi. Takto vygenerovaného jedince ukládám do pole *d_population* umístěnou v globální paměti. Pole s populací je ve formátu jednorozměrného pole a jednotliví jedinci jsou uloženi za sebou.

Jelikož se pohybujeme na GPU, kde je podpora paralelizace, tak funkce pro generování populace generuje všechny jedince populace najednou. To je umožněno pomocí indexace na obrázku níže. Každé vlákno pracuje nad daty právě jednoho svého jedince.

```
int tidx = blockDim.x * blockIdx.x + threadIdx.x;
```

Výpis kódu 2: Indexace paralelizace

7.1.2 Ohodnocení fitness (findFitnessCostFlowshop())

Jedná se o stěžejní funkci, která vypočítá kvalitu všech jedinců v nově vygenerovaném poli *d_population*. Tato funkce vypočítá fitness pro každého jedince v populaci. Podrobný popis jak algoritmus funguje je již popsáno v teoretické části této práce a již ho nebudu opakovat. Pouze se zaměřím na důležité části kódu při implementaci. Vytvořil jsem si pomocné pole *lastJob*, která vždy obsahuje aktuální dobu provádění úkolů na jednotlivých strojích. Proměnná *flowshopI* obsahuje vždy index elementu v poli se zadáním, s kterým se bude pracovat. Bylo potřeba si vytvořit i pravdivostní proměnnou *prvniJob*, která mi ověřuje, jestli se provádí výpočet prvního úkolu na prvním stroji.

```
__global__ void findFitnessCostFlowshop(int *d_population, int *d_flowshop,
int *d_fitness)
{
	int i = blockDim.x * blockIdx.x + threadIdx.x; //jedinec z populace
budu inkrementovat
	int ii = i * DIMENSION;

	if (i < DIMENSION * POPSIZE)
	{
	int lastJob[MACHINECOUNT] = { 0 };
	bool prvniJob = true;

	for (int j = 0; j < DIMENSION; j++) //pro vsechny joby
	{
		int flowshopI = d_population[ii] - 1;

		for (int k = 0; k < MACHINECOUNT; k++) //pro vsechny machiny
		{
			if (prvniJob == true)
			{
				if (k == 0) //pokud je prvni tak nescitat PRO
PROVNI MACHINU
				{
					lastJob[k] = d_flowshop[flowshopI];
					flowshopI += DIMENSION;
				}
				else
				{
					lastJob[k] = d_flowshop[flowshopI] +
lastJob[k - 1];
					flowshopI += DIMENSION;
				}
			}
			else //pro ostani JOBY
			{
				if (k == 0) // PRO PROVNI MACHINU
				{
					lastJob[k] += d_flowshop[flowshopI];
					flowshopI += DIMENSION;
				}
				else
				{
					//najdi maximum a pricti flow[x]
					lastJob[k] = __max(lastJob[k - 1],
lastJob[k]) + d_flowshop[flowshopI];
					flowshopI += DIMENSION;
				}
			}
			if (k == MACHINECOUNT - 1)
			{
				prvniJob = false;
			}
		}
		ii++;
	}
	d_fitness[i] = lastJob[MACHINECOUNT - 1];
	} //endIF
}
```

Výpis kódu 3: Implementace hledání fitness

Tato funkce vyžaduje pole *d_flowshop*, která obsahuje zadání Flow Shop problému, které se snažíme vyřešit. Pole jsem nejprve musel vytvořit v paměti RAM počítače a až následovně překopírovat do globální paměti grafické karty. To jsem provedl pomocí funkce *cudaMemcpy()*, která je součásti knihovny CUDA. Nesmíme zapomenou i na alokování dostatečného místa před kopírováním v paměti grafické karty. To se provede funkcí *cudaMalloc().*

```
int *d_flowshop;
cudaMalloc((void**)&d_flowshop, DIMENSION * MACHINECOUNT *
sizeof(int));
cudaMemcpy(d_flowshop, flowshop, DIMENSION * MACHINECOUNT *
sizeof(int), cudaMemcpyHostToDevice);
```

Výpis kódu 4: Kopírování Flow Shop zadání

Na obrázku výše vidíte implementaci, která kopírujeme pole *flowshop* obsahující zadání problému do pole *d_flowshop* umístěné v paměti grafické karty. Třetím parametrem metody se určuje, kolik dat chceme překopírovat a další parametr určuje, kterým směrem kopírování chceme provést. Možnost *cudaMemcpyHostToDevice* zajišťuje kopírování z paměti RAM do paměti GPU. Důvod, proč jsem nevytvořil pole rovnou v kernelu funkce je jednoduchý. Pole se zadáním může být obrovské, a pokud bych to udělal touto cestou, každý kernel by vytvořil svoji vlastní kopii zadání Flow Shop. To by byla obrovská zátěž na paměť a mým řešením je pole v paměti grafické karty kopírovat pouze jednou. Nevýhodou je, že kopírování mezi pamětí počítače a grafické karty není efektivní, ale toto kopírování provádím jen jednou a nemá tak obrovský dopad na výkon.

Na obrázku implementace si můžete povšimnout, že implementace myslí na paralelizaci a výpočet pro všechny jedince populace probíhá současně. To je zajištěno správnou indexací. Každé vlákno ukládá výsledek výpočtu na správný index pole *d_fitness*, která obsahuje všechny výsledky výpočtu v paměti grafické karty.

7.1.3 Hledání vůdce v populaci (findLeader())

Zde jsem využil jednoduchý přístup hledání minima v poli. Pole *d_fitness*, které obsahuje všechny ohodnocení fitness všech jedinců, jsem rozdělil na menší intervaly. Pro každý menší interval jsem spustil funkci *findLeader()*. Ta je navržena tak, že sekvenčně prochází krátkou část pole a postupně do proměnné *best* ukládá dosud nejlepšího nalezeného jedince. Další volání funkce ještě vyhledá minimum z pole mezivýsledků získaných z prvního volání funkce. Funkce ukládá do pole s názvem *d_leader* elementy nejlepšího nalezeného jedince.

7.1.4 Výpočet pole J (calculateJ())

Tato funkce vytváří pole *J*, pomocí kterého budeme později hledat nejčastější obsažený prvek. Podrobně je to popsáno v teoretické části. Odečítám odpovídající element vůdce od elementu jiného jedince. To vše pod absolutní hodnotou, aby nevznikaly záporná čísla. Výpočet absolutní hodnoty zajistila vestavěná funkce *abs()*. Výsledky postupně ukládám do nového pole *d_Jj*. Funkce je opět jednoduše paralelizovatelná a každé vlákno se postará o výpočet jednoho jedince, tak jak je ukázáno v obrázku níže.

```
__global__ void calculateJ(int *d_population, int
*d_leader, int *d_Jj){
      int i = blockDim.x * blockIdx.x +
threadIdx.x;
      int ii = i * DIMENSION;

      if (ii == 0 || ii % DIMENSION == 0){
            for (int j = 0; j < DIMENSION; j++){
                  d_Jj[ii] = abs(d_leader[j] -
d_population[ii]);
                  ii++;
            }
      `
```

Výpis kódu 5: Vytváření pole J

7.1.5 Hledání nejčastějšího prvku v poli J (jHistogram())

Dalším krokem bylo zjistit, které číslo se nejčastěji vyskytuje v poli *J*. Zde jsem si vytvořil pole představující histogram. Kde index pole bude představovat, o jaký prvek se jedná a samotná hodnota v poli bude jeho četnost v poli *J*. Toto pole s názvem *d_histo* jsem si nejprve vytvořil v globální paměti grafické karty a bylo potřeba ho nastavit tak, aby všechny jeho hodnoty byly z počátku nuly. K tomu jsem si vytvořil funkci *jHistogramReset()*. Tato funkce je opět paralelizovaná tak, že každý index pole nuluje jiné vlákno. Další funkce *jHistogram()* vytváří samotný histogram četnosti výskytu prvků.

```
__global__ void jHistogram(int* hst,
int* data)
{
      int x = blockDim.x * blockIdx.x +
threadIdx.x;

      if (x < POPSIZE * DIMENSION)
      {
            int idx = data[x];
            atomicAdd(&(hst[idx]), 1);
      }
```

Výpis kódu 6: Vytvoření histogramu

Na obrázku výše můžete vidět reálnou implementaci, kde jsem využil atomické operace *atomicAdd()*. Tato funkce inkrementuje proměnnou či pozici v poli o proměnnou, kterou zadáme jako druhý parametr. Tato funkce má obrovskou výhodu a podporuje paralelní přístup a ošetřuje konflikt přístupu více vláken do jedné části paměti. To znamená, pokud dojde k situaci, kdy se budou chtít do jedné části paměti zapisovat dvě vlákna, tak se s inkrementací počká, dokud se nedokončí inkrementace vláknem, které přistoupilo dříve. Funkci jsem samozřejmě paralelizoval a každý převod prvku z pole J do histogramu provádí jiné vlákno.

Experimentoval jsem i s využitím sdílené paměti, kam jsem si ukládal částečné výpočty, pro akceleraci výpočtu, ale žádná implementace mi nevylepšila rychlost výpočtu. V mém případě se základní ukázala jako nejefektivnější. Jedná se o optimalizovanou funkci.

7.1.6 Výpočet délky kroku s (calculatesFromJmax())

Pro tuto část kódu nebylo potřeba nijak zdlouhavě vymýšlet nějakou implementaci založenou na paralelizaci. A to je zapříčiněno tím, že zde pouze hledáme nejčastější prvek v histogramu *d_histo*, který uložím do proměnné J_{max}. Navíc tento histogram obsahuje velice málo prvků. Při testování se budeme pohybovat v počtech desítek a maximálně ve stovkách prvků. Dále ještě musíme vypočítat hodnotu proměnné *s*. Přesný postup je vysvětlen v teoretické části v sekci výpočet délky skoku. I tato část funkce je puštěna pouze v jednom vláknu, protože zde hledáme právě jednu společnou hodnotu, která bude totožná pro všechny jedince v populaci. Výslednou hodnotu délky kroku s, uložím do globální paměti grafické karty. Zde se proměnná jmenuje *d_s*. CUDA obsahuje v současné verzi kvalitní knihovnu s matematickými operacemi. Při implementaci jsem potřeboval pro výpočet vzorce *s* převést vypočtenou hodnotu, která byla ve formátu reálného čísla, na hodnotu obsahující celé číslo. Navíc zde bylo omezení, že se hodnota vždy zaokrouhlovala dolů. K tomu jsem využil vestavěnou matematickou funkci *floor()*, která zaokrouhlení provedla za mě.

7.1.7 Vytvoření skokové matice G (calculateG())

Tato funkce vytváří nové skokové jedince a to za pomoci algoritmu pro výpočet matice *G* z teoretické části. Z jednoho jedince populace vznikne J_{min} nových jedinců, takže matice *G* bude velice obsáhlá. Účelem této části kódu je vytvořit nové jedince a vynulovat element jedince z populace, pokud jsou rovny s prvkem vůdce populace, tak jak je uvedeno ve vzorci pro výpočet. Při implementaci jsem narazil na potíže, kde se mi vytvářeli noví skokoví jedinci, kteří měli nevhodné elementy. To je nepřístupné, protože jedinec může obsahovat pouze elementy v množině od 1 do velikosti dimenze. Algoritmus jsem tedy upravil, a pokud došlo k tomuto jevu, rovnou jsem element nastavil na hodnotu 0. Nulové stavy se řeší v pozdější části algoritmu DSOMA a jsou ji označeny nedosažitelné stavy.

```
for (int l = 1; l <= Jmin; l++) // 1 - Jmin //Jmin
      { //Jmin
            int counter = (ii / (DIMENSION * Jmin)) * DIMENSION;
            for (int j = 0; j < DIMENSION; j++){
                  if (d_population[counter] < d_leader[j]){
                        d_G[ii] = d_population[counter] + *d_s * l;
                              if (d_G[ii] >= DIMENSION || d_G[ii] <= 0)
                                    {
                                          d_G[ii] = 0;
                                    }
                                    ii++;
                                    counter++;
                              }
                              else if (d_population[counter] >
d_leader[j])
                              {
                                    d_G[ii] = d_population[counter] -
*d_s * l;
                                    if (d_G[ii] >= DIMENSION || d_G[ii]
<= 0)
                                    {
                                          d_G[ii] = 0;
                                    }
                                    ii++;
                                    counter++;
                              }
                              else {
                                    d_G[ii] = 0;
                                    ii++;
                                    counter++;
                              }

                        }
                  }
```

Výpis kódu 7: Výpočet matice G s modifikací

Proměnná *counter* vždy obsahuje index elementu jedince v populaci, s kterým se pracuje a index *ii* určuje, kam v rámci pole *G* se výsledek výpočtu ukládá. Všichni nově vytvoření jedinci jsou vytvořeni v jednom poli nazvaném *d_G* a nejprve jsou za sebou uspořádané všechny skoky jednoho jedince a až poté jedince následujícího, až do počtu velikosti populace. V tomto případě byla využita paralelizace pro výpočet každého jedince populace zvlášť, kde v rámci jednoho vlákna proběhl kompletní výpočet všech nových skokových jedinců.

7.1.8 Vytváření pokusných jedinců (constructingTrial())

Tato funkce přetransformuje množinu skokových jedinců uložených v poli *d_G* na množinu pokusných jedinců uložených v poli *d_H* v globální paměti grafické karty. Podrobný popis algoritmu je v teoretické části vytváření pokusných jedinců. Bylo potřeba překontrolovat každého jedince z *G*. Pokud jedinec na různých elementech obsahoval více shodných hodnot, tak se druhý a následující element nastavil na nulu. Toto chování jsem naimplementoval pomocí využití pomocného pole *buffer*, kde jsem si postupně ukládal všechny unikátní hodnoty jedince. K výpočtu mi dopomohla logická proměnná *isUnique*, která kontrolovala, jestli při průchodu polem jedince je stále prvek unikátní. Pokud ano tak se prvek zkopíruje do pole *buffer*. Pokud však prvek již v populaci je obsažen a logická proměnná je nastavena na *false*, tak se hodnota elementu nastaví na nulu. Postup bylo potřeba naprogramovat sekvenčně, aby se prohledal celý jedinec.

Konstrukce se paralelizovala velice jednoduše. Výpočet každého skokového jedince zajišťovaly různá vlákna. Implementace paralelizace je ukázána na obrázku níže.

```
int idx = blockDim.x * blockIdx.x + threadIdx.x;
int ii = idx * DIMENSION * Jmin;
```

Výpis kódu 8: Paralelizace výpočtu pokusných jedinců

7.1.9 Oprava pokusných jedinců (repairTrial())

V této funkci provádím opravy pokusných jedinců vytvořených v předchozí funkci pro vytvoření testovacích jedinců. Úkolem této části algoritmu je nahradit nulové elementy jedince hodnotami, které ještě nejsou obsaženy v jedinci. Nulou jsou indexovány všechny nedostupné stavy, které je potřeba opravit. Postup opravy je implementován následovně. Nejprve je potřeba zjistit, které prvky jedince chybí. Lépe řečeno, nalezneme rozdíl množiny H. Sekvenčně procházím jedince v *d_H* od 1 do velikosti dimenze jedince a do pole *Hdiff* si uložím všechny prvky, které nejsou obsaženy v jedinci. K tomu mi dopomohla logická proměnná *IsInIndiv*, které mi sledovala, jestli byl element nalezen v jedinci. V dalším kroku bylo potřeba tyto nalezené prvky promíchat. Na to jsem vytvořil třídící algoritmus, který postupně projde pole a každý prvek prohodí s prvkem náhodně vygenerovaným pomocí funkce *curand()*, která zajišťuje generování náhodných čísel v CUDA. Takto promíchané pole prvků, které chybí v jedinci, uložím do společného pole nazvaného *d_diff* uloženého v globální paměti grafické karty.

Další funkce nazvaná *repairTrial2()* vkládá již do elementů jedince indexovaných nulou sekvenčně patřičné hodnoty z pole *d_diff*. Proměnná *Hindex* označuje index elementu v poli *d_H* a *diffCounter2* označuje index v poli *d_diff*.

```
int Hdiff[DIMENSION] = { 0 };
int diffcount = 0;
bool isInIndiv = false;

//rozdil mnoziny H
for (int i = 1; i <= DIMENSION; i++)
      {
            for (int j = ii * DIMENSION; j < ii * DIMENSION +
DIMENSION; j++)
            {
                  if (i == H[j])
                  {
                        isInIndiv = isInIndiv || true;
                  }
            }
            if (isInIndiv == false)
            {
                  Hdiff[diffcount] = i;
                  diffcount++;
            }
            else
                  isInIndiv = false;
      }
```

Výpis kódu 9: Ukázka nalezení rozdílu jedince v H

Obě funkce zajišťující opravy jsou paralelizované. Výpočty nad jedinci pokusných jedinců běží paralelně na různých vláknech.

7.1.10 Aktualizace populace (populationUpdate())

Finální procedura, která musí být provedena nad testovací populace je ověření, jestli nově vytvoření jedinci v poli *d_H* vylepšili jedince populace, z kterého vznikli tito noví jedinci. K tomu bude potřeba znovu spustit funkci *findFitnessCostFlowshop()*, kterou jsme již spouštěli jednou na začátku algoritmu nad počáteční populací. Tentokrát však spustíme výpočet ohodnocení fitness nad polem *d_H*, který obsahuje všechny testovací jedince.

```
dim3 nbThreadsFFH(128);
dim3 nbBlocksFFH(((POPSIZE * Jmin) / nbThreadsFFH.x) + 1);
findFitnessCostFlowshopH << < nbBlocksFFH, nbThreadsFFH >>
>(d H, d flowshop, d fitnessH);
```

Výpis kódu 10: Hledání fitness na poli d_H

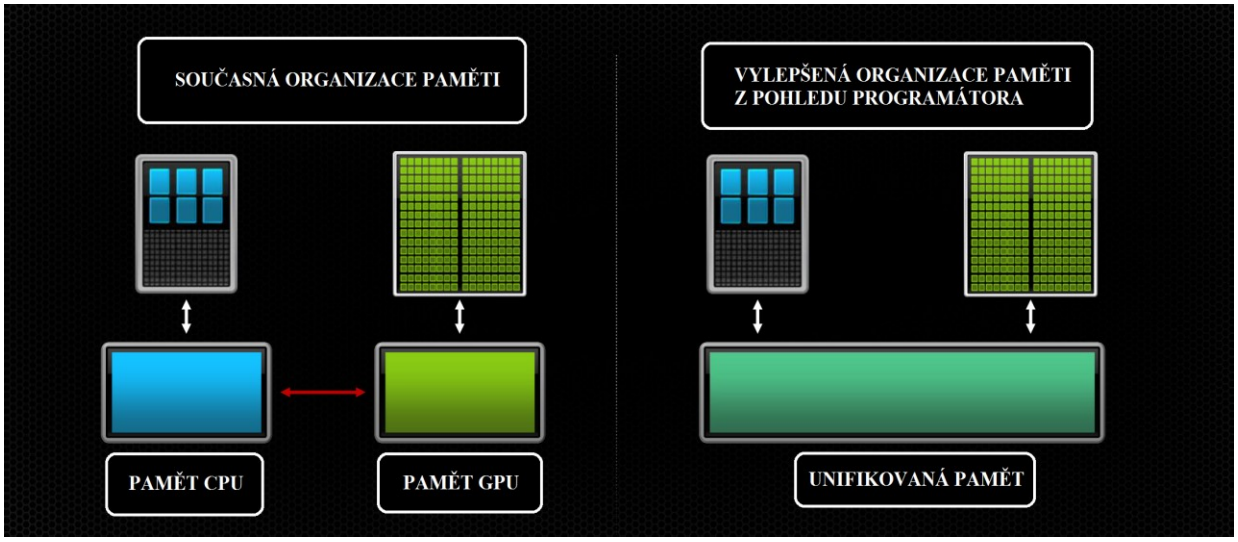
Výsledné ohodnocení jedinců ukládám do pole *d_fitnessH*. Nyní už nic nebrání porovnat jedince s jeho testovacími jedinci. Sekvenčně porovnám všechny testovací jedince patřičného jedince v populaci, a pokud se algoritmem DSOMA vylepšil v hodnotě fitness, nastavím jej jako nového jedince populace. Tento test provádím paralelně pro každého jedince populace.

7.1.11 Nalezení nejlepšího jedince v populaci (minFit())

Naimplementoval jsem si pomocnou funkci, která mi nalezne nejlepšího jedince v populaci, který představuje nejlepší nalezené řešení našeho problému. Funkce projde fitness ohodnocení celé populace a najde nejlepšího jedince s nejmenší hodnotou fitness.

7.2 Experimenty s unifikovanou pamětí

V rámci diplomové práce jsem se měl zaměřit na unifikovanou paměť. V této kapitole vysvětlím, proč jsem toto vylepšení nevyužil. Unifikovaná paměť byla vydána jako součást CUDA ve verzi 6.0. A hlavním účelem přidání tohoto vylepšení bylo ulehčení práce s pamětí pro programátory. Pro ilustraci uvádím oficiální obrázek této architektury.



Obrázek 4: Unifikovaná paměť (zdroj: devblogs.nvidia.com)

Na oficiálním obrázku vývojářů CUDA můžete vidět, jak tento zjednodušený model vypadá. Paměť počítače má společnou paměť s grafickou kartou. Je to však vytvořeno pouze na virtuální úrovni. Takže pomoci tohoto modelu nemusíme provádět explicitně kopírování mezi pamětmi, ale je to uděláno automaticky. Po mém testování a porovnání s klasickým modelem jsem od tohoto nápadu upustil.

Jelikož v rámci mého projektu provádím kopírování pouze dvakrát. A to na začátku programu při kopírování zadání do grafické paměti a na konci programu, kde kopíruji jen výsledné nejlepší řešení ztratil tento model smysl a nepřináší žádné výkonností zlepšení. Zjednodušuje však paměťový model, takže můžeme mít veškerá data uložená na jednom místě.

To se změní již brzy a společnost NVIDIA vydá vylepšení NVLink. Jedná se o vylepšení, které přináší hardwarový vysokorychlostní tunel, skrz který bude paměť počítače a grafické karty komunikovat.

7.3 Praktická implementace v C# a ověření funkčnosti

Pro účely testování jsem vytvořil velice podobnou kopii algoritmu DSOMA i v jazyce C#. Nemá smysl znovu popisovat jednotlivé metody algoritmu, pouze vypíšu odlišnosti ve zdrojovém kódu.

Rozdíl v implementaci nebyl až tak velký. Zde jsem využil také paralelismu s využitím funkce paralelní smyčky for, kterou jsem namapoval na jedince populace. Aplikace tedy využívá při běhu všechna dostupná jádra procesoru. Protože výpočet probíhá kompletně na procesoru, nebylo potřeba data nikam přesouvat a odpadla alokace paměti mimo paměť počítače. Populaci jsem vygeneroval do dvojrozměrného pole. Na rozdíl od druhého algoritmu, kde veškeré výpočty probíhaly nad jednorozměrným polem. Dalším rozdílem bylo vytváření histogramu pro výpočet *J*. Použil jsem tabulku hash, do které jsem postupně ukládal počty jednotlivých prvků. Ostatní prvky algoritmu jsou již naprogramovány stejným způsobem.

Po naimplementování všech potřebných funkcí bylo potřeba ověřit, zda je vše funkční podle zadání. Test funkčnosti jsem provedl na nejmenším testovaném zadání Flow Shop s 5 úkoly na 4 strojích. Vždy jsem využil malé populace několika jedinců a kontroloval funkčnost funkcí. U aplikace v C# bylo ladění aplikace jednodušší. Příklad je natolik jednoduchý, že lze dopočítat a ověřovat vizuálně. Vždy jsem si nechal vypisovat mezivýsledky jednotlivých funkcí a ověřoval jejich správnost. Navíc vývojové prostředí Visual Studio mě s testováním velice pomohlo a zjednodušilo tím, že mi zobrazovalo mezivýsledky funkcí.

Ověřování funkčnosti CUDA verze aplikace bylo o hodně složitější, protože zde bylo využíváno paralelismu a vývojové prostředí si s ním vždy neporadilo a neumožnilo zobrazit mezivýsledky funkcí. Musel jsem tedy pro jistotu jednotlivé mezivýsledky zkopírovat z paměti grafické karty do RAM, vypsat si je na standartní výstup a překontrolovat, případně přepočítat.

7.4 Spouštěcí parametry kernelu a vytížení multiprocesoru grafické karty

Parametry spouštění jednotlivých kernelů jsou ukázány na výpisu kódu 10. Všechny výše popsané kernely jsem spouštěl ve 128 vláknech v rámci jednoho bloku. Počet spuštěných bloků se dynamicky měnil podle složitosti problému a hlavně podle velikosti populace. Z důvodu ověření dobré implementace algoritmu DSOMA a co největšího vytížení grafické karty bylo potřeba provést test vytížení multiprocesoru grafické karty. K tomu společnost NVIDIA vytvořila kalkulátor vytíženosti CUDA [11]. Jedná se o předpřipravený formulář, kde stačí pouze doplnit vstupní parametry a jsou zobrazeny reálné výsledky vytížení multiprocesoru grafické karty. Povinnými vstupními parametry je zadání správné verze Compute Capability, která v mém případě je 3.5. Dále musíme zadat v kolika vláknech v rámci bloku kernel pouštím. Všechny kernely pouštím ve 128 vláknech. Posledním parametrem je počet využitých registrů v rámci jednoho vlákna. K získání této informace bylo potřeba spustit DSOMA algoritmus se speciálním parametrem *–ptxas-options=-v*, který mi potřebnou informaci zobrazil pro všechny spuštěné kernely. Využití se pohybovalo v rozmezí 3 až 31 registrů napříč všemi mými kernely. Výsledky měření jsou následující a jsou totožné pro všechny mé spuštěné kernely. Počet aktivních vláken v rámci multiprocesoru je 2048. Počet bloků spuštěných v rámci multiprocesoru je 16. Vytížení využitého multiprocesoru je 100%. Aplikaci jsem ověřil a skutečně využívá velkého potenciálu grafické karty. Důkazem může být i pozdější test vytíženosti počítače.

8. Metodika testování

V této kapitole je popsáno, jak bude probíhat testování výkonu, měření škálovatelnosti a ověřování kvality výsledků DSOMA algoritmu. Proběhne taky porovnání kvality a rychlosti výpočtu algoritmu napsaného v jazyce C# pro procesor a v CUDA určeného pro grafickou kartu. Metodiku jsem rozdělil do dvou podkapitol, kde se napřed zaměřím na porovnání obou implementovaných algoritmů a nakonec se zaměřím pouze na podrobnější testy na CUDA aplikaci.

8.1 Porovnávací testy

8.1.1 Porovnání obou algoritmů

Druhým testem bude ověření výkonu algoritmu DSOMA vytvořeného v CUDA s DSOMA algoritmem napsaným v jazyce C#. Test proběhne na malém příkladu Flow Shop problému. Konkrétně se jedna o Flow Shop problém na pěti strojích s dvaceti úkoly. Testovací příklad je převzat ze stránek Taillard Benchmark [3], kde jsou volně ke stažení zadání různě velkých Flow Shop problémů. Jsou zde i k vidění dosud nejlepší získané výsledky.

Také se zaměřím na škálování výkonu obou algoritmů. Postupně budu zvyšovat parametry jako POPSIZE, Jmin a MIGRATIONS a v tabulce budu porovnávat, jak si jednotlivé algoritmy stojí.

U obou algoritmů budu porovnávat dobu výpočtu a uvedu je v poměru o kolik je CUDA algoritmus rychlejší.

Spouštěcí parametry jsou následující:

- MIGRATIONS – 10
- DIMENSION – 20
- POPSIZE – 1000
- Jmin – 1

8.1.2 Zaměnitelnost velikost populace a migrace

Standardně při řešení evolučních algoritmu se využívá relativně malé populace jedinců. Jejich počet se pohybuje do 1000 jedinců. Naopak se provádí velký počet migračních cyklů samotného evolučního algoritmu. DSOMA se však při implementaci ukázal jako velice sériový algoritmus a efektivní paralelizace se dala vytvořit pouze v rámci celých jedinců. To znamená, že provádění funkcí v rámci jedince zajišťovalo jediné vlákno.

Úkolem tohoto testu je ověřit možnost zaměnitelnosti větší populace za provedení menšího počtu migračních cyklů a malé populace při provedení většího počtu migračních cyklů. Faktory, které nás budou zajímat je rychlost výpočtu a kvalita dosaženého řešení. Parametry pro spuštění velké populace a malého počtu migrací je následující

- MIGRATIONS – 100
- DIMENSION – 20
- POPSIZE – 10 000
- Jmin – 1

Parametry pro spuštění druhé varianty budou opačné:

- MIGRATIONS – 10 000
- DIMENSION – 20
- POPSIZE – 100
- Jmin – 1

Testování bude probíhat na zadání ta011, které je součásti Taillard testovací sady příkladů a jedná se o příklad 20 úkolů na 10 strojích. Měření pro každý případ zopakuji pětkrát, abych ověřil stabilitu kvality výsledků. Uvedu i průměrnou hodnotu C_{max}, a průměrnou dobu výpočtu. Pro srovnání provedu testy na obou mých aplikacích, aby bylo vidět porovnání rychlosti výpočtu a kvality řešení.

8.2 Podrobné CUDA testy

8.2.1 Efektivita kódu napsán pro CUDA

Prvním testem se bude ověřovat, jak dlouho trvá výpočet jednotlivých částí kódu DSOMA algoritmu v CUDA. V tabulce vypíšu, jak jsou které části algoritmu výpočetně náročné. Výsledky časové náročnosti výpočtu jsou uvedeny v milisekundách. Výsledné hodnoty budu udávat v přesnosti na desetiny milisekund.

Měření probíhalo při tomto nastavení:

- MIGRATIONS – 1
- DIMENSION – 20
- POPSIZE – 1000
- Jmin – 1

8.2.2 Test výkonu nad různě velkými problémy Flow Shop

Další část testování již probíhá pouze na algoritmu DSOMA vytvořeném pro CUDA. Úkolem je nalézt co nejkvalitnější řešení dané úlohy, při použití dostatečně velkých hodnot spouštěcích parametrů. Tento výsledek porovnám s dosud nejlepším výsledkem, který kdy byl nalezen pro danou úlohu. K tomuto testování využiju opět Taillard Benchmark úkoly a vždy uvedu, o které zadání se jedná. U každé úlohy je vždy nejlepší řešení, které bylo doposud nalezeno. Mé řešení s touto nejlepší hodnotou budu porovnávat podle vzorce 8.1 uvedeného níže.

$$Avg = \frac{(M - B) * 100}{B} \tag{8.1}$$

Výsledek vzorce Avg naznačuje, jak moc kvalitního řešení jsem dosáhnul. Snaha je se dostat co nejblíže k hodnotě 1. Algoritmus DSOMA spustím vždy 5x a vytvořím tabulku s průměrem výsledných hodnot. Takto ověřím, jak často DSOMA produkuje kvalitní řešení. Vypíšu také nejlepší výsledek měření, pro porovnání. U každého testu budou vždy dvě tabulky. První bude obsahovat nejlepší řešení C_{max}, kterého jsem dosáhl v jednotlivých testech a čas výpočtu. Tabulka bude obsahovat i průměr naměřených hodnot. Druhá tabulka bude obsahovat globálně nejlepší řešení problému, které bylo kdy nalezeno, mé nejlepší řešení a pomocí vzorce výše vypočtena kvalita DSOMA řešení.

Test proběhne nad problémy velikosti:

- 5 úkolů a 4 stroje
- 20 úkolů a 5 strojů
- 20 úkolů a 10 strojů
- 50 úkolů a 20 strojů
- 100 úkolů a 5 strojů

Spouštěcí parametry budou následující.

- MIGRATIONS – 1000
- DIMENSION – mění se podle velikosti úkolu (5 - 100)
- POPSIZE – 10 000
- Jmin – 1

8.2.3 Snaha o nalezení úplně nejlepšího řešení

V tomto testu se již zaměříme na nalezení co nejlepšího řešení Flow Shop problému za využití vhodného nastavení parametrů. Testovacím příkladem bude problém s 20 úkoly na 5 strojích Taillard Benchmark testovací sady. Nastavení jsem zvolil po zkušenostech z předešlých měření s ohledem na rychlost algoritmu a kvality výsledku. Nejlepší mé nalezené řešení opět porovnám s řešením nejlepším pomocí rovnice pro Avg a zhodnotím výsledky.

Nastavení parametrů bude následující.

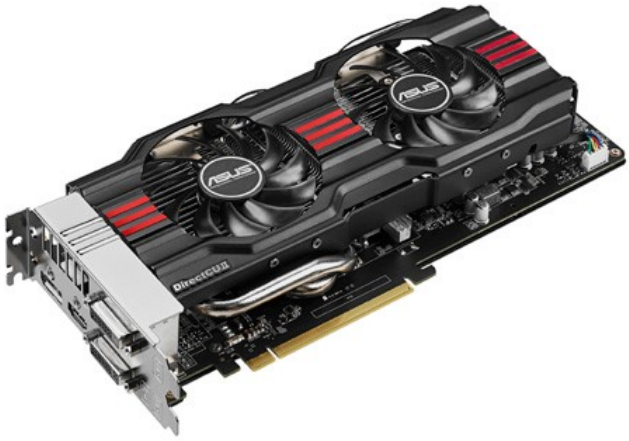
- MIGRATIONS – 1 000 000
- DIMENSION – 20
- POPSIZE – 10 000
- Jmin – 1

8.2.4 Test využití prostředků počítače

V závěrečném testu se podrobněji podíváme, jak jsou využity prostředky počítače při spuštěném CUDA aplikace. Bude nás zajímat zatížení procesoru, zatížení grafických jader, frekvence na grafických jádrech a pamětích a využití grafické paměti. Pro toto testování jsem použil aplikaci MSI Afterburner, které mi umožňuje ladit parametry grafické karty a dokáže monitorovat využití výše popsaných prostředků. Pro měření využití prostředků procesoru jsem použil vestavěnou funkci ve Windows 10 s názvem Správce úloh. Z těchto aplikací ukážu vygenerované grafy a popíšu je.

9. Testování

V této kapitole proběhne testování podle předem uvedené metodiky. Ukážeme si i počítačovou sestavu pro testy. Podrobný popis počítačové sestavy, na které proběhlo veškeré testování, naleznete na obrázku níže. Experimenty probíhaly na produktech střední třídy, kde jejich cena, která je také velkým faktorem, byla téměř srovnatelná. Grafická karta stála 8000 Kč a procesor 6000 Kč. To je třeba při zhodnocení výsledků zohlednit. Při dlouhodobém provádění výpočtů se nesmí zapomenout na odběr elektrické energie. Obě aplikace jsem proměřil wattmetrem a při běhu C# program byla spotřeba 120 W. Při běhu CUDA programu se spotřeba vyšplhala na 210 W.



Grafický čip	GK104
Počet CUDA jader	1536
Paměť GDDR5	2048 MB
Šířka sběrnice	256 b
Propustnost sběrnice	227 GB/s
Frekvence GPU	1167 MHz
Frekvence paměti	7100 MHz

Obrázek 5: Vlastnosti GPU a obrázek zpracování (zdroj: alza.cz)

Komponenty:

- Procesor: Intel Core i5 2500k @ 4,2GHz
- Základní deska: ASUS P8P67-PRO
- Paměti: 2 x 4GB DDR3 Crucial @ 1333MHz
- Grafická karta: ASUS GeForce GTX 770 DirectCUII
- SSD disk: Samsung SSD 850 EVO 250 GB

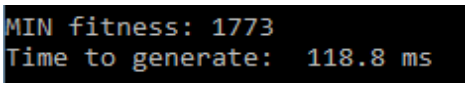
Nainstalované programy nutné k provedení implementace algoritmu DSOMA a jejímu testování. Veškeré testování probíhalo na operačním systému Microsoft Windows 10 v poslední verzi 1511.

Další nutné programy jsou následující:

- Vývojové prostředí: Visual Studio 2013 Professional
- Ovladač grafické karty: GeForce Game Ready Driver 364.72
- Sada nástrojů CUDA: CUDA Toolkit v poslední verzi 7.5
- Propojovací nástroj CUDA s vývojovým prostředím: Nsight Visual Studio Edition 4.0
- Program pro monitorování zatížení GPU: MSI Afterburner 4.2

9.1 Efektivita kódu napsaném pro CUDA

Zde je tabulka 4 s výsledky časové náročnosti výpočtu jednotlivých funkcí. Měření probíhalo tak, že jsem před spuštěním DSOMA algoritmu nastartoval časovač. Tento časovač jsem po provedení potřebné funkce zastavil a nechal jsem si vypsat výsledek. Na obrázku 6 lze vidět, jak jsem v konzoli zobrazoval výsledná data.



Obrázek 6: Ukázka vypsání výsledku

Výsledky měření provedené na DSOMA algoritmu vytvořeném v CUDA.

Tabulka 4: Výsledky měření časové náročnosti výpočtu

FUNKCE	ČAS VÝPOČTU
Alokace paměti na GPU a kopírování zadání	0,3 ms
Generování populace	9,9 ms
Výpočet fitness	0,3 ms
Hledání vůdce	0,1 ms
Výpočet J	0,4 ms
Nulování histogramu	0,1 ms
Naplnění histogramu	0,1 ms
Výpočet s	0,1 ms
Generování G	0,4 ms
Vytváření pokusných jedinců	0,1 ms
Oprava jedinců 1	1,8 ms
Oprava jedinců 2	0,1 ms
Výpočet fitness pokusných jedinců	0,3 ms
Aktualizace populace	0,1 ms
CELKEM	14,7 ms

Z výsledků je patrné, že nejdéle trvá výpočet počáteční populace a to 9,9 ms. Je to způsobeno naplňováním velkého pole hodnotami, která jsou generována pomocí náhodného generátoru dat za určitých podmínek. Tato funkce se však provádí pouze jednou na začátku algoritmu DSOMA a při

navýšení počtu migračních cyklů už nehraje žádnou roli. Dalším náročnějším kusem kódu byla Oprava jedinců 1, která trvala 1,8 ms. Zde se hledá množina elementů, která není v rámci jednoho jedince a hlavně zde probíhá promíchání této množiny. Promíchání je založeno na funkci *curand()*, která brzdí tuto operaci. Výpočty fitness se prováděly 0,3 ms. To je způsobeno zejména velikostí zadání Flow Shop. Zadání má 20 úkolů prováděných na 10 strojích. Ostatní procedury již zabíraly zanedbatelnou dobu výpočtu vůči výše popsaným funkcím.

9.2 Porovnání obou algoritmů

V následujícím testu se podíváme na výsledky měření výkonu DSOMA aplikací na obou platformách. V přehledné tabulce také ukážu, jak se výkon mezi sebou liší. Tabulka vždy v prvním sloupci označuje, na kterém parametru provádíme pokusy. Ostatní jsou v základním nastavení, tak jak jsou uvedeny v metodice testování. V druhém sloupci je uvedena doba výpočtu algoritmu v CUDA. Naopak ve třetím sloupci jsou výsledky měření výkonu pro C# verzi algoritmu DSOMA. Obě tyto položky jsou v hodnotách milisekund.

9.2.1 Testy algoritmů

V první části tohoto testu se zaměříme na velikost populace. Ukážeme si, jak se mění výkon v závislosti na velikosti vygenerované populace.

Tabulka 5: Porovnání algoritmů na velikosti populace

POPULACE	CUDA	C#	CUDA lepší
100	30	15	0,5
200	33	20	0,6
500	34	30	0,9
1000	37	45	1,2
2000	42	80	1,9
3000	44	111	2,5
5000	51	178	3,5
10 000	86	346	4,0
20 000	152	671	4,4

Z tabulky vidíte, že již pro populaci 1000 jedinců je na tom CUDA aplikace 1,2x lépe. Výpočet CUDA aplikace trval 37 ms a C# aplikace 45 ms. Při populaci od 100 do 500 jedinců byla CUDA aplikace ještě pomalejší. To je způsobeno zejména malým využitím paralelizace. Jednoduše při výpočtech nad malým počtem jedinců využijeme velice malou část výpočetního výkonu grafické karty. Pro větší populace však CUDA přejímá vedení. Populace 5000 jedinců již předběhne C# aplikaci více jak 3,5x a tento trend neustává. Zvyšování velikosti populace se ukázal jako klíčový parametr, kde CUDA aplikace získává obrovský náskok.

Druhá část testu ověřuje závislost výkonu při zvyšování počtu skoku Jmin.

Tabulka 6: Porovnání algoritmů na velikosti skoku

Jmin	CUDA	C#	CUDA lepší
1	37	45	1,2
2	39	60	1,5
3	42	72	1,7

Z tabulky je patrné, že opět již pro minimální hodnotu Jmin = 1 SOMA napsána v CUDA je 1,2x rychlejší oproti referenční verzi pro procesor. V CUDA výpočet trval 37 ms a v C# verzi aplikace 45 ms. Další zvyšování parametru Jmin ještě více škáloval výkon. Pro 3 skokové sekvence již byl výkon více jak 1,7x rychlejší. Zde však nemá smysl parametr nastavovat na vyšší hodnotu.

Poslední parametr, na kterém jsem ověřil výkon obou algoritmů, byl počet provedených migračních cyklů.

Tabulka 7: Porovnání algoritmu na počtu migrací

MIGRATION	CUDA	C#	CUDA lepší
1	15	15	1
10	37	45	1,2
100	260	333	1.2
200	485	648	1,3

Tabulka jasně ukazuje, že zvyšováním počtu migračních cyklů vyšší výkon CUDA aplikace nezískáme. Při 10 migračních cyklech je CUDA aplikace sice 1,2x rychlejší, ale tento náskok je způsoben obecně vyšším výkonem CUDA. Pro větší počet migrací již výkon stoupal velice pomalu. Již před měřením jsem měl tušení, že zvyšováním tohoto parametru výkon navíc nezískám, protože se jedná o sériový výpočet algoritmu SOMA a paralelizace zde nepomůže.

9.2.2 Souhrnné výsledky

Z předešlých měření vyšlo najevo, že pro nízké hodnoty parametrů jako jsou populace 1000 jedinců a menší a délku skoku 1 je náskok CUDA aplikace malý, ale již je rozpoznatelný. To však není pravda pro vysoké hodnoty parametrů. Konkrétně se jedná o populaci 5000 jedinců a 2 skoků Jmin. Ostatní hodnoty jsou nastaveny na základ uvedený v metodice testování.

Tabulka 8: Výsledky kombinovaného měření pro populaci a počet skoků 2

POPULACE	CUDA	C#	CUDA lepší
5000	72	253	3,5
20 000	205	969	4,7

Tabulka ukazuje, že pro toto nastavení je výpočet na grafické kartě již 3,5x rychlejší a to není zanedbatelné zrychlení. Výpočet na grafické kartě trval 72 ms a C# výpočet 253 ms. Pro vyšší nastavení jednotlivých parametrů se výkon jednotlivých aplikací dále prohluboval.

9.3 Test výkonu nad různě velkými problémy Flow Shop

Zde uvedu výsledky testování nad různě velkými problémy Flow Shop podle metodiky pro testování uvedené v předešlé kapitole práce. Vždy uvedu přehlednou tabulku s výsledky měření a pokusím se o zhodnocení a popsání naměřených hodnot.

9.3.1 Flow Shop 5 úkolů a 4 stroje

Jedná se o jednoduchý příklad, na kterém jsme si popisovali způsob řešení Flow Shop problému v teoretické části práce, který není součástí Taillard měřících testů. Zde ještě není potřeba ukazovat tabulku s výsledky měření, protože se jedná o natolik jednoduchá příklad, který má pouze 120 různých permutací a nejlepší řešení je C_{max} = 32, kterého dosahuje několik různých permutací. Algoritmus DSOMA si s ním poradí již při první migraci, proto není potřeba cokoliv měřit.

9.3.2 Flow Shop 20 úkolů a 5 strojů

Prvním složitějším testem je Flow Shop s 20 úkoly na 5 strojích. Test se jmenuje ta001 a ukážu, jak si algoritmus DSOMA s testem poradil. Tento test má již 20! řešení a výpočet hrubou silou by trval už opravdu dlouho.

Tabulka 9: Naměřené hodnoty Flow Shop 20 x 5

Test	C_{max} min	Čas
1.	1297	6s
2.	1324	6s
3.	1335	6s
4.	1339	6s
5.	1332	6s
Průměr	1325	6s

Vypočtené hodnoty nejlepších C_{max} jsou v intervalu 1297 až 1339, kde průměrná hodnota byla 1325. Výpočet algoritmu trval pouze 6 s. Pomocí velké náhody za využití DSOMA jsem nalezl opravdu dobré jedno řešení C_{max} = 1297. Další již byly průměrné a pohybovaly se v úzkém intervalu.

Tabulka 10: Porovnání výsledku Flows Shop 20 x 5

Nejlepší řešení	1278
Nejlepší mé C_{max}	1297
Avg	1,5

Nejlepší mé řešení mělo hodnotu 1297. Dosud nejlepší nalezené řešení je 1278. Na tyto údaje jsem použil vzorec pro výpočet Avg a vyšla hodnota 1,5. Jak můžete vidět, výsledek je již velice blízko nejlepšímu řešení a dá se považovat za kvalitní.

9.3.3 Flow Shop 20 úkolů a 10 strojů

Výsledky měření pro test ta011 Taillard testovacího zadání jsou následující.

Tabulka 11: Naměřené hodnoty Flow Shop 20 x 10

Test	C_{max} min	Čas
1.	1719	6s
2.	1734	6s
3.	1730	6s
4.	1713	6s
5.	1727	6s
Průměr	1725	6s

Můžete vidět, že nejlepší řešení, kterého jsem v pěti testech dosáhl, bylo C_{max} = 1713. Hodnoty jednotlivých testů se pohybovaly v rozmezí od 1713 do 1734. Doba výpočtu byla 6 s a u všech měření byla konstantní. Výpočet tedy trval stejnou dobu jako předchozí Flow Shop problém s 5 stroji. Průměrné řešení bylo 1725.

Tabulka 12: Porovnání výsledku Flows Shop 20 x 10

Nejlepší řešení	1582
Nejlepší mé C_{max}	1713
Avg	8,3

Druhá tabulka ukazuje porovnání nejlepšího řešení s řešením mým. Pro tento příklad dosud nejlepší řešení je 1582. Jak už bylo zmíněno, mé nejlepší řešení je 1713. Po výpočtu vzorce pro Avg vyšla hodnota kvality řešení 8,3. Tato hodnota je ještě docela vysoká a nalezené řešení ještě není moc kvalitní. V takto krátkém čase se nepodařilo nalézt lepší řešení.

9.3.4 Flow Shop 50 úkolů a 20 strojů

Dalším testem s ještě větším počtem potencionálních řešení je test ta051 s 50! možnými řešeními.

Tabulka 13: Naměřené hodnoty Flow Shop 50 x 20

Test	C_{max} min	Čas
1.	4414	25s
2.	4380	25s
3.	4433	25s
4.	4424	25s
5.	4420	25s
Průměr	4414	25s

Z tabulky můžete vyčíst, že nejlepší nalezené řešení mělo hodnotu C_{max} = 4380. Naopak nejhorší řešení mělo hodnotu 4433. Průměr byl 4414. Tyto výsledky byly zjištěny za 25 sekund. To už je čtyřnásobný čas oproti problému s 20 úkoly.

Tabulka 14: Porovnání výsledku Flows Shop 50 x 20

Nejlepší řešení	3875
Nejlepší mé C_{max}	4380
Avg	13

Nejlepší nalezený výsledek má minimální C_{max} = 3875. Řešení nalezené algoritmem DSOMA má hodnotu 4380. Potom $Avg = 13$. Řešení považuji za špatné, protože jsem stále daleko od ideálního výsledku a Avg to dokázal. Zde se projevil jev, kde DSOMA algoritmus získá mnohem horší řešení u problému na větším počtu strojů. K nalezení lepšího řešení je potřeba mnohem více migračních cyklů.

9.3.5 Flow Shop 100 úkolů a 5 strojů

Posledním testem z Taillard testovací sady je ta061. Zadání má už obrovské množství možných řešení a to 100!.

Tabulka 15: Naměřené hodnoty Flow Shop 100 x 5

Test	C_{max} min	Čas
1.	5588	85s
2.	5609	85s
3.	5592	85s
4.	5610	85s
5.	5527	85s
Průměr	5585	85s

Výpočet takto velkého testovacího příkladu trval 85 sekund. Vypočtené hodnoty nejlepších C_{max} se pohybovaly od 5527 do 5610. Průměrná hodnota činila 5585.

Tabulka 16: Porovnání výsledku Flow Shop 100 x 5

Nejlepší řešení	5493
Nejlepší mé C_{max}	5527
Avg	0,6

Obecné nejlepší řešení má hodnotu 5493. Mé implementaci algoritmu DSOMA se podařilo nalézt nejlepší výsledek 5527. To znamená, že po využití vzorce pro výpočet Avg jsem získal hodnotu 0,6. To považuji za velice kvalitní řešení již po tak malém počtu migrací.

9.3.6 Zhodnocení výsledků

Již po provedení 1000 migrací algoritmus vykázal stabilní výsledky výpočtu. Vypočtené nejlepší hodnoty se pohybovaly ve velice malém intervalu. Projevila se zajímavá vlastnost algoritmu DSOMA. Testy, které proběhly na Flow Shop problémech s menším počtem strojů, vykazovaly daleko lepší kvalitu nalezených řešení. Pro porovnání problém 20 úkolů s 5 stroji měl hodnotu kvality výsledku $Avg = 1{,}5$ a test se stejným počtem úkolů ale na dvojnásobném počtu strojů vykázal hodnotu $Avg = 8{,}3$. To jasně ukazuje na větší sílu algoritmu pro menší počet strojů.

9.4 Snaha o nalezení úplně nejlepšího řešení

Test probíhal na testovacím příkladů sady Taillard s názvem ta001. Jedná se o Flow Shop problém s 20 úkoly na 5 strojích. Po provedení velkého počtu migračních cyklů se dalo očekávat kvalitní řešení problému.

Tabulka 17: Porovnání výsledku Flow Shop 20 x 5 na dlouhém testu

Nejlepší řešení	1278
Nejlepší mé C_{max}	1297
Avg	1,5
Čas výpočtu	115 min
Pořadí provedení úkolů	9, 15, 1, 3, 17, 19, 13, 11, 8, 14, 6, 18, 16, 4, 5, 2, 10, 7, 20, 12

V tabulce již vidíte výsledky provedeného testu. Podařilo se mi nalézt řešení s C_{max}= 1297. V porovnání s nejlepším řešením jsem se už pohyboval velice blízko optimálního řešení, které je 1278. Kvalitu obou řešení jsem opět porovnal díky vzorci pro výpočet Avg. Výsledkem je hodnota 1,5 a jedná se už o velice kvalitní řešení blížící se optimálnímu řešení. V tomto případě jsem uvedl kompletní řešení, které DSOMA algoritmus získá a vypsal jsem pořadí provedení úkolů pro C_{max} = 1297.

9.5 Zaměnitelnost velikost populace a migrace

Prvním testem bylo ověření rychlosti a kvality výsledků při využití 10 000 jedinců populace při provedení 100 migračních cyklů. Testování probíhalo na příkladu ta011 z Taillard testovací sady příkladů. V tabulce 18 můžete vidět přehledně seřazené výsledky měření. Pro CUDA aplikaci naměřené C_{max} se pohybovalo v rozmezí 1706 do 1738. Průměrná hodnota potom byla 1726. Čas výpočtu byl téměř konstantní a jeho průměr byl 677 ms. Pro C# algoritmus byl již výpočet o poznání pomalejší a trval 3,2 s. Průměrná minimální hodnota byla o trochu vyšší a činila 1742.

Tabulka 18: Naměřené hodnoty velké populace 10 000 jedinců a 100 migrací

	CUDA		C#	
Test	C_{max} min	Čas	C_{max} min	Čas
1.	1735	692 ms	1766	3,2 s
2.	1706	671 ms	1771	3,2 s
3.	1735	685 ms	1753	3,2 s
4.	1738	666 ms	1735	3,2 s
5.	1718	672 ms	1712	3,2 s
Průměr	1726	677 ms	1747	3,2 s

Tabulka 19: Naměřené hodnoty malé populace 100 jedinců a 10 000 migrací

	CUDA		C#	
Test	C_{max} min	Čas	C_{max} min	Čas
1.	1768	16 s	1805	3,7 s
2.	1753	16 s	1824	3,7 s
3.	1798	16 s	1820	3,7 s
4.	1766	16 s	1775	3,7 s
5.	1779	16 s	1821	3,7 s
Průměr	1772	16 s	1809	3,7 s

V tabulce 19 již můžete vidět naměřené hodnoty pro druhý případ testu. To je varianta při použití velice malé populace 100 jedinců a provedení 10 000 migračních cyklů. Zde již CUDA aplikace na první pohled vykazovala horší výsledky měření. C_{max} se pohybuje od 1753 do 1798 a průměrná hodnota tedy činí 1772. Čas výpočtu je v porovnání s prvním testovaným případem o hodně horší a trval 16 s. C# aplikace si v tomto případě vedla o hodně lépe. Pro malou populaci výpočet trval pouze 3,7 s a to je více jak 4x rychlejší než konkurenční aplikace. Průměrná kvalita řešení byla trochu horší a činila 1809.

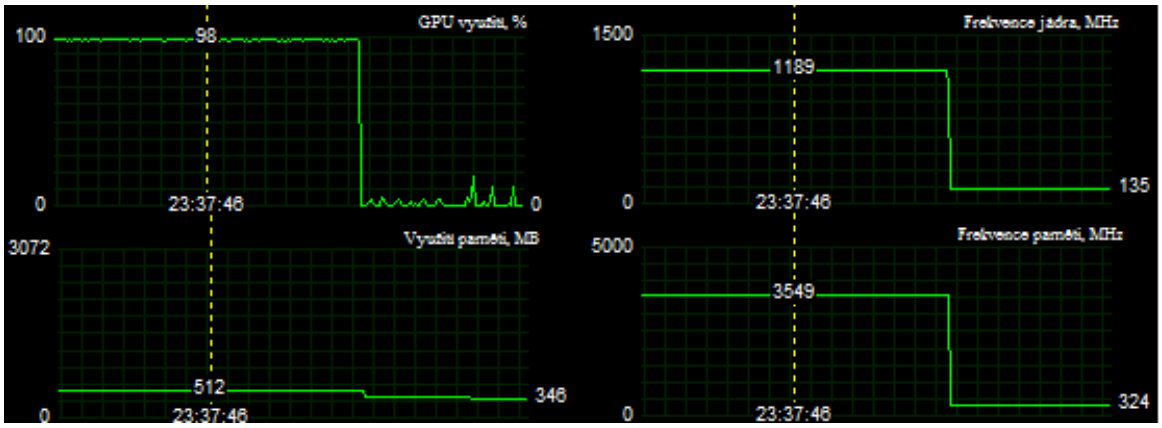
9.5.1 Zhodnocení výsledků zaměnitelnosti

Výsledky testování jsou jednoznačné. Pro CUDA DSOMA spuštěná na malém počtu jedinců při velkém počtu migračních cyklů není stejně efektivní jako druhý měřený případ. Získané výsledné hodnoty C_{max} jasně ukazují, že pro větší populaci získáme kvalitnější řešení již po pár migračních cyklech. Rozdíl je průkazný a změřitelný. Také rychlost výpočtu jasně ukazuje na vyšší počet jedinců, kdy paralelní přístup

aplikace provede výpočet mnohonásobně rychleji. Dalším zjištěním je, že C# algoritmus je pro malou populaci daleko vhodnější. Výpočet proběhl více jak 4x rychleji. Je to způsobeno zejména minimálním využitím prostředků grafické karty u CUDA algoritmu, kde využijeme jen zlomek výpočetních jader. Výsledky měření si můžete porovnat na tabulkách 18 a 19. Podobné výsledky jsem očekával a je to způsobeno zejména velikosti problému. Testovaný případ má 20! možných řešení a algoritmus DSOMA z velké míry ovlivňuje faktor náhodných čísel. Osvědčila se mi však varianta velké populace a vysokého množství migračních cyklů, kde již DSOMA vykazuje solidních výsledků.

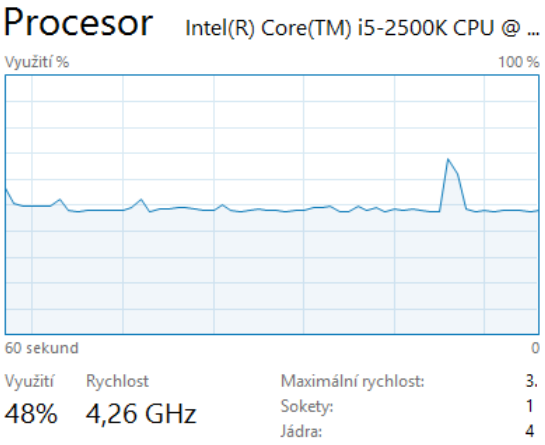
9.6 Využití prostředků počítače

Kapitola obsahuje výsledky měření využití prostředků počítače za pomoci programu MSI Afterburner a Správce úloh. Dále ukážu snímky obrazovky, které reprezentují reálné využití prostředků, které jsem pořídil při testování pomocí výše uvedené aplikace.



Obrázek 7:Využití grafické karty

Na prvním obrázku vidíte využití grafické karty (GPU usage), která jasně ukazuje, že algoritmus DSOMA v CUDA využil většinu jejího výpočetního výkonu a držel se na hodnotě 98%. U využití paměti grafické karty (Memory usage) bylo potřeba odečíst základní hodnotu paměti 345 MB. Testy probíhaly na operačním systému Windows 10 a tyto prostředky si pro sebe alokoval samotný systém. Finální využití grafické paměti algoritmem DSOMA potom činí 167 MB. Poslední grafy dokazují, že grafická karta běžela ve výkonnostním režimu a frekvence jader (Core clock) byla mnou nastavených 1189 MHz. To platí i pro frekvenci paměti (Memory clock), kde však vidíte pouze 3549 MHz. Jedná se o paměti GDDR5, kde musíme výslednou frekvenci vynásobit dvěma, abychom se dostali na hodnotu efektivní. Potom se frekvence paměti vyšplhala na 7098 MHz.



Obrázek 8:Využití procesoru

Druhý obrázek ukazuje zajímavý vedlejší účinek při běhu CUDA programu. Využití procesoru bylo 48%, což není zanedbatelné zatížení procesoru. To by mělo být způsobeno čekáním procesoru na provedení jednotlivých kernelů. Kernely běží v sérii za sebou a výpočet dalšího musí proběhnout až po provedení předchozího. Můžete vidět i frekvenci procesoru, kterou jsem nastavil na hodnotu 4,26 GHz.

9.7 Konečné zhodnocení výsledků

Při měření efektivity DSOMA v CUDA jsem dosáhl očekávaných výsledků. Metody, které byly komplexnější, byly vypočítávány nejdelší dobu. Také metody využívající vestavěnou funkci pro generování náhodných čísel byly velice náročné na výpočet.

Při porovnání algoritmů SOMA v CUDA a C# byly pro malé vstupní parametry výsledky podobné. Pro populaci do 1000 jedinců byla C# aplikace dokonce rychlejší. Avšak při zvyšování hodnot parametrů začal SOMA algoritmus škálovat. Již pro populaci 5000 jedinců a parametr $J_{min} = 2$ byl program naimplementovaný v CUDA 3,5x rychlejší. Tento trend se stále zvětšoval pro vyšší hodnoty parametrů.

Zaměřil jsem se také na zaměnitelnost velikosti populace s počtem provedení migračních cyklů. Obecný přístup evolučních algoritmů je provádět výpočty nad relativně malou populací a provést velký počet migrací. Já se však pokusil experimentovat a spustil jsem algoritmus DSOMA i nad větší populací do 20 000 jedinců. Důvodem byl dostatek výpočetních prostředků pro toto testování ze strany grafické karty. Experiment skutečně vykázal lepší výsledky pro velkou populaci a malý počet migrací, kde výpočet na CUDA trval velmi krátkou dobu v porovnání s C#. Výsledné řešení bylo taky prokazatelně lepší. Důvodem tohoto chování je zejména minimálního využití paralelizace, protože pro malé populace využívám jen zlomek výpočetních jader a problém se nerozloží na celou grafickou kartu.

Při testování různě velkých zadání Flow Shop problémů jsem ověřil, že již po menším počtu migrací algoritmus dosahoval slušných výsledků. Hodnota Avg se pohybovala v rozmezí 1 až 13. To jsou slušná řešení pro výpočet trvající méně než 2 minuty. Nejlepších výsledků jsem dosáhl u problémů s menším počtem strojů pro libovolný počet úkolů. Ukázala se i vysoká stabilita kvality výsledků. Výsledná řešení byla v malém intervalu. Navíc se mi podařilo nalézt i velmi kvalitní řešení, která vyčnívala z nalezeného průměru, v takto krátkém čase.

Pokusil jsem se i nalézt co nejkvalitnější řešení Flow Shop problému. Parametry jsem nastavil co nejefektivněji, aby algoritmus provedl 1 000 000 migračních cyklů. Výsledná hodnota $Avg = 1{,}5$ byla již velice blízko nejlepšímu řešení. Na ideální řešení však nedosáhla. Výpočet takto velkého množství migračních cyklů trval velice krátkou dobu a to 115 minut.

10. Závěr

Tématem této diplomové práce byla implementace DSOMA algoritmu v CUDA a porovnat jej s referenčním řešením vytvořeným v jazyce C#. Porovnání mělo proběhnout na permutačně optimalizačních problémech. Měl jsem se zaměřit zejména na rychlost běhu jednotlivých algoritmů a škálovatelnost řešení.

Nejprve jsem si však musel nastudovat danou problematiku, kterou jsem podrobně popsal v teoretické části na začátku práce. Napřed jsem objasnil problematiku optimalizačních problémů a popsal, že se nejedná o triviální problém vyřešit tyto úlohy. Další kapitola již vysvětlovala, co je to plánování a proč se v praxi vyplatí plánovat určité procesy a co tím získáme. Uvedl jsem i nejznámější plánovací úlohy, které se využívají v počítačových vědách. Popsal jsem stručně, co to je problematika Job Shop. Podrobně jsem se zaměřil na objasnění Flow Shop problému, který je speciálním případem Job Shop problému. Důkladně jsem popsal danou problematiku a na jednoduchém příkladu jsem ukázal, jak vyřešit tuto úlohu, abych znalosti mohl využít v praktické části práce.

Dále jsem se již zaměřil na vysvětlení heuristických algoritmů, protože v rámci práce jsem měl implementovat evoluční algoritmus, který je založen na tomto modelu. V kapitole evolučních algoritmů jsem uvedl jejich obecné vlastnosti.

Následující kapitola se věnovala velice důležitému tématu práce a to byla technologie CUDA. Implementace algoritmu probíhala v CUDA a bylo potřeba se podrobně seznámit s touto architekturou. Popsal jsem historii, vývoj a hlavní výhody, které programování na grafické kartě přináší. Zaměřil jsem se i na vysvětlení paralelizace.

Posledním tématem teoretické části práce je samotný algoritmus SOMA a zejména jeho diskrétní verze. Stručně jsem popsal základní algoritmus a zaměřil se na podrobné vysvětlení modifikované verze nazvané DSOMA. Popsal jsem jednotlivé procedury algoritmu, abych znalosti mohl využít při praktické implementaci.

První kapitola praktické části se již věnuje implementaci DSOMA v CUDA. U jednotlivých procedur jsem názorně ukázal, jak jsem implementaci provedl a pokud bylo potřeba, tak jsem vysvětlil, proč jsem zvolil tento přístup. Kapitola vysvětluje i unifikovanou paměť. Tento paměťový model však nepřinesl žádný výkonnostní přínos, tak jsem od jeho využití v implementaci upustil. Potom stručně popisuji algoritmus CUDA vytvořený v jazyce C#. Zaměřil jsem se pouze na odlišnosti v porovnání s programem v CUDA.

Další kapitola obsahuje metodiku pro testování výkonu DSOMA. Popsal jsem způsob měření efektivity CUDA aplikace, porovnání obou implementovaných aplikací, test zaměnitelnosti velikosti populace a počtu migrací, test výpočtu nad různě velkými problémy Flow Shop, pokus o nalezení co nejlepšího řešení a metodiku, jak byly využity prostředky počítače při výpočtech. Uvedl jsem také, jaká byla testovací počítačová sestava a co bylo potřeba mít nainstalováno pro bezproblémovou implementaci a testování.

Poslední kapitola se již čistě věnuje výsledkům testování, které jsem provedl podle vytvořené metodiky pro testování. Vždy jsem v tabulce ukázal získaná data a popsal jsem je. U každého testu jsem provedl zhodnocení.

Vývoj v oblasti grafických karet a výpočtech na nich jde velkou rychlostí kupředu a výkonnostně pro vhodné úlohy dalece předběhl klasický výpočet na procesoru. Tato oblast mě velice zajímá a jsem rád, že jsem si tuto skutečnost mohl ověřit na evolučním algoritmu DSOMA, který dosahuje výborných výsledků na této platformě. Jsem opravdu zvědav, jak poroste výkon grafických karet v blízké budoucnosti a co vše na nich bude možno implementovat, aby programy získaly výsledky složitých výpočetních problémů v reálném čase.

11. Literatura

[1] Boyd, Stephen P.; Vandenberghe, Lieven (2004). Convex Optimization. Cambridge University Press. p. 129. [cit. 15. Dubna 2016] Dostupné na http://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf

[2] Blazewicz, J., Ecker, K.H., Pesch, E., Schmidt, G. und J. Weglarz, Scheduling Computer and Manufacturing Processes, Berlin (Springer) 2001 [cit. 15. Dubna 2016]

[3] Taillard, E. (January 1993). "Benchmarks for basic scheduling problems". European Journal of Operational Research 64: 278–285. [cit. 15. Dubna 2016] Dostupné na http://mistic.heig-vd.ch/taillard/

[4] Garey, M. R., Johnson, D. S., & Sethi, R. (1976). The complexity of flowshop and jobshop scheduling. Mathematics of operations research. [cit. 15. Dubna 2016]

[5] NVIDIA Corporation. NVIDIA CUDA, Santa Clara, 2016. [cit. 15. Dubna 2016] Dostupná na http://www.nvidia.com/object/cuda_home_new.html

[6] I. Zelinka, Prezentace do Biologicky Inspirovaných výpočtů, 2012 [cit. 15. Dubna 2016]

[7] D. Davendra, Disertační práce Evoluční Algoritmy - DSOMA, 2011 [cit. 15. Dubna 2016]

[8] Pearl, Judea (1983). Heuristics: Intelligent Search Strategies for Computer Problem Solving. New York, Addison-Wesley, p. vii. [cit. 7. Července 2016]

[9] Gareth Jones, Genetic and Evolutionary Algorithms, University of Sheffield, UK, 2000 [cit. 7. Července 2016]

[10] NVIDIA Corporation. NVIDIA CUDA Docs, Santa Clara, 2016. [cit. 15. Dubna 2016] Dostupná na http://www.docs.cuda.com

[11] NVIDIA Corporation. CUDA Occupancy Calculator, 2016. [cit. 15. Dubna 2016] Dostupná na http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

12. Příloha

Algoritmus DSOMA v CUDA
Algoritmus DSOMA v C#